Deep Learning Part 1

By Jordi Cathew

Table of Contents

1. Neural Network Regression Notes..... .4
2. Neural Network Classification Notes..... .8
   2.1. Activation Functions..... .9
   2.2. Learning rates..... .12
   2.3. Classification evaluation Methods..... .16
3. Computer Vision and Convolutional Neural Networks..... .18
   3.1. Scaling (normalizing) images..... .20
   3.2. An in-depth explanation of CNNs..... .23
   3.3. Continuing notes..... .32
   3.4. Improving a model from a data perspective..... .34
4. Transfer Learning..... .35
   4.1. Feature extraction..... .36
   4.2. Fine-Tuning..... .36
   4.3. Scaling up..... .42
5. NLP (Natural Language Processing)..... .46
   5.1. NLP Datasets..... .49
   5.2. Converting Text Into Numbers in NLP Problems..... .50
      5.2.1. Tokenization in TensorFlow..... .52
      5.2.2. Embedding in TensorFlow..... .54
   5.3. Model 0 (baseline) – Naive Bayes with TF-IDF encoder..... .55
   5.4. Model 1: Feed-Forward Network (Dense)..... .57
      5.4.1. Visualizing Embeddings..... .59
   5.5. RNN's (Recurrent neural networks)..... .60
      5.5.1. Unfolding RNNs..... .65
      5.5.2. Sequence Modelling: Design Criteria..... .66
      5.5.3. Backpropagation Through Time (BPTT)..... .66
      5.5.4. Gradient Problems..... .69
      5.5.5. LSTMs (Long Short Term Memory Networks)..... .72
      5.5.6. Variants of LSTMs..... .78
      5.5.7. Limitations of RNNs and How to solve them?..... .86
   5.6. Model 2: LSTM (RNN)..... .90
   5.7. Model 3: GRU (RNN)..... .91
   5.8. Model 4: Bidirectional LSTM (RNN)..... .92

5.9. CNNs for Sequence Problems..... .95
5.9.1. Inner workings..... .96
5.10. Model 5: Conv1D..... .97
5.11. Model 6: Using Transfer Learning (Pretrained Sentence Encoder).....98
5.12. Model 7: Same as model 6 but with 10% of the data..... .100
5.13. Attention..... .101
5.13.1. Vanilla Attention..... .105
5.14. Transformers..... .106
5.14.1. History..... .106
5.14.2. Multi-Head Attention..... .110
5.14.3. Transformer-based GPT example..... .112

1. Neural Network Regression Notes

Compiling a Model: After defining the model architecture and before starting the training process, we need to compile the model. Compilation involves specifying additional configurations that are necessary for the training process. These configurations include the optimizer, loss function, and optional metrics.

Fitting a Model: When we fit a model, we train it on a given dataset by iteratively updating its parameters based on the input data and the corresponding target values. The primary objective of fitting is to optimize the model's parameters to minimize the chosen loss function and improve its predictive performance. Compiling a Model: After defining the model architecture and before starting the training process, we need to compile the model. Compilation involves specifying additional configurations that are necessary for the training process. These configurations include the optimizer, loss function, and optional metrics.

---

*Question: If we need to touch some hyperparameters of a neural network (*Parameters are the configuration model, which are internal to the model. Hyperparameters are the explicitly specified parameters that control the training process*), how do we choose which? Of course we could try changing the epochs, the learning rate, etc one by one, but this seems inefficient, and very slow, eventually we will hit the nail yes, but we spent too much time and we don't even know what was causing the loss in our model.*

Answer:

When it comes to optimizing the parameters of a neural network, it's important to have a systematic approach rather than randomly changing parameters one by one. Here are some strategies you can use to efficiently tune the parameters and identify the factors affecting the model's performance:

1. **Define a Search Space**: Start by defining a reasonable range or set of values for the parameters you want to optimize. This includes parameters like learning rate, batch size, number of hidden layers, number of neurons per layer, regularization strength, etc. Consider the possible values that are relevant to your problem and limit the search space.
2. **Use Grid Search or Random Search**: Grid search and random search are popular techniques for **hyperparameter optimization**. Grid search involves defining a grid of parameter combinations and exhaustively evaluating the model's performance for each combination. Random search, on the other hand, randomly samples parameter values from the defined search space. These methods allow you to systematically explore different parameter combinations and evaluate their impact on the model's performance.
3. **Cross-Validation**: Utilize cross-validation techniques such as k-fold cross-validation to evaluate the model's performance for each parameter setting. Split your training data into multiple folds, train the model on a subset of folds, and evaluate it on the remaining fold. This helps in obtaining more reliable performance metrics and reduces the risk of overfitting to a specific dataset.
4. **Automated Hyperparameter Optimization**: Consider using automated tools or libraries for hyperparameter optimization, such as scikit-learn's `GridSearchCV` or `RandomizedSearchCV`, or specialized libraries like Optuna or Keras Tuner. These tools automate the process of searching for optimal hyperparameters by performing efficient search algorithms and providing performance evaluation.
5. **Track Performance and Metrics**: Keep track of the model's performance metrics, such as accuracy, loss, or other relevant evaluation metrics, for each parameter

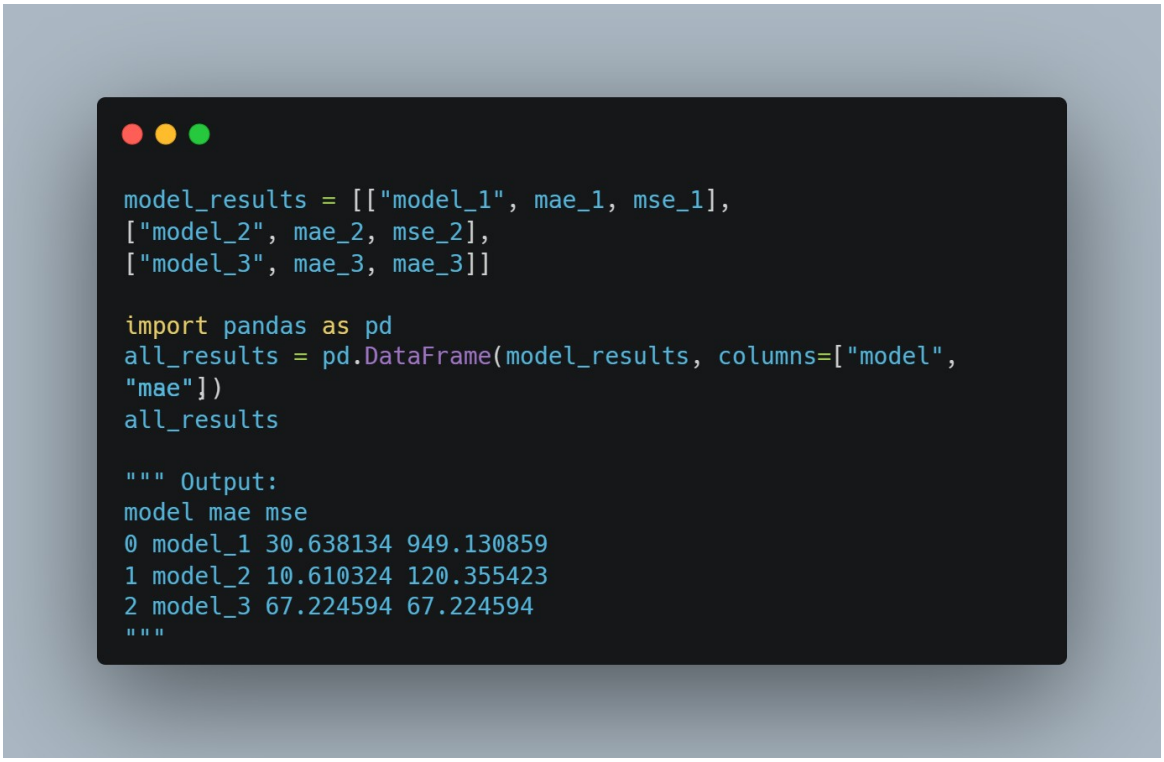
setting. This allows you to compare the performance of different configurations and identify the combinations that yield the best results.

6. **Analyze and Iterate**: Analyze the results and identify the parameters that have the most significant impact on the model's performance. Use the knowledge gained from the previous iterations to guide your subsequent parameter adjustments. You can perform further fine-tuning by narrowing down the search space or adjusting the step sizes for continuous parameters.
7. **Domain Knowledge and Prior Experience**: Understanding the nature of the problem and the characteristics of the data can help you make more intelligent choices in optimizing the parameters.

It's important to note that parameter optimization is an iterative process, and it may require multiple rounds of experimentation and adjustment. By following a systematic approach and using techniques like grid search, random search, cross-validation, and automated hyperparameter optimization, you can efficiently explore the parameter space and identify the factors that contribute to the model's performance.

Additionally, it's worth noting that there are advanced techniques like Bayesian optimization and genetic algorithms that can further optimize hyperparameters by intelligently sampling the parameter space based on previous results. These techniques can be explored once you have a good understanding of the problem and have exhausted simpler approaches.

Comparing results of different models using pandas:



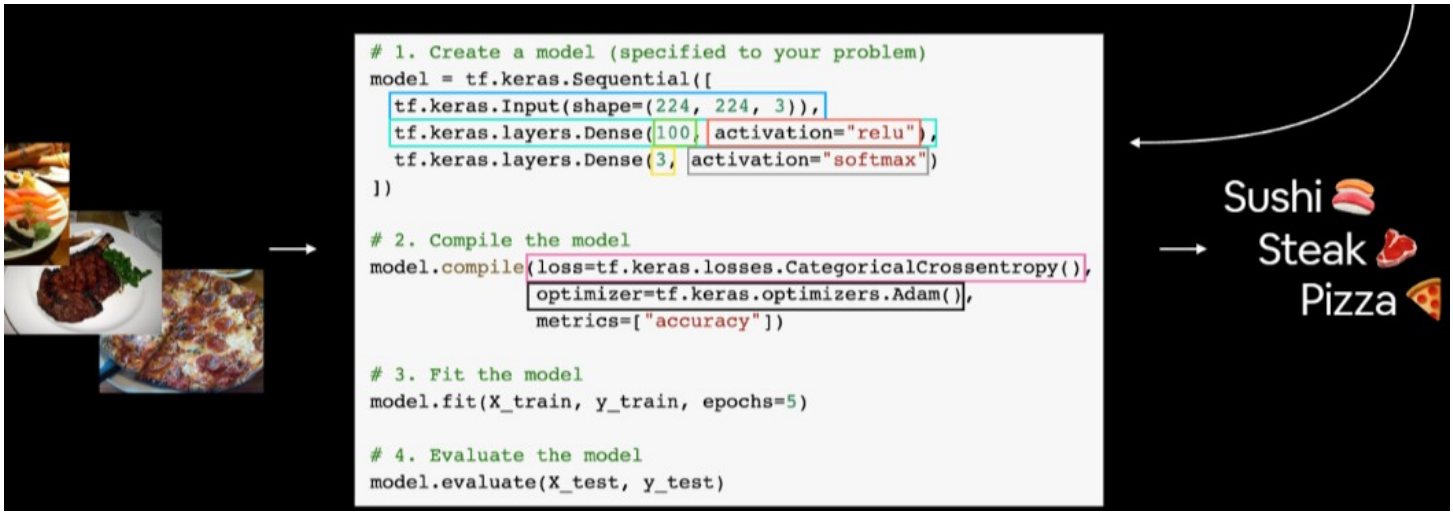
```
model_results = [["model_1", mae_1, mse_1],
["model_2", mae_2, mse_2],
["model_3", mae_3, mae_3]]

import pandas as pd
all_results = pd.DataFrame(model_results, columns=["model",
"mae"])
all_results

""" Output:
model mae mse
0 model_1 30.638134 949.130859
1 model_2 10.610324 120.355423
2 model_3 67.224594 67.224594
"""
```

2. Neural Network Classification Notes

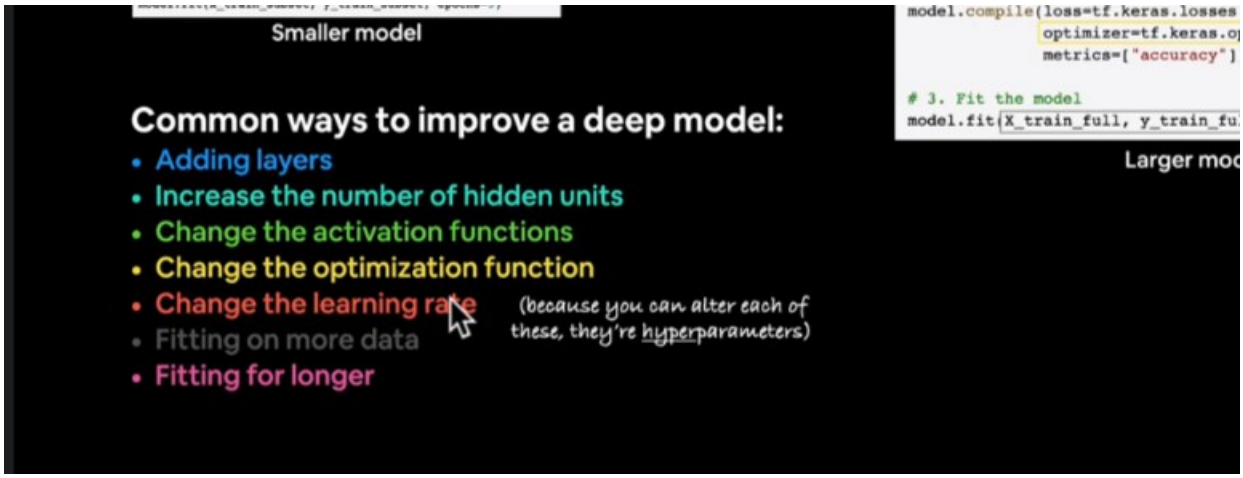
Architecture of a classification model:

Hyperparameter	Binary Classification	Multiclass classification
Input layer shape	Same as number of features (e.g. 5 for age, sex, height, weight, smoking status in heart disease prediction)	Same as binary classification
Hidden layer(s)	Problem specific, minimum = 1, maximum = unlimited	Same as binary classification
Neurons per hidden layer	Problem specific, generally 10 to 100	Same as binary classification
Output layer shape	1 (one class or the other)	1 per class (e.g. 3 for food, person or dog photo)
Hidden activation	Usually ReLU (rectified linear unit)	Same as binary classification
Output activation	Sigmoid	Softmax
Loss function	Cross entropy (`tf.keras.losses.BinaryCrossentropy` in TensorFlow)	Cross entropy (`tf.keras.losses.CategoricalCrossentropy` in TensorFlow)
Optimizer	SGD (stochastic gradient descent), Adam	Same as binary classification

Source: Adapted from page 295 of Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow Book by Aurélien Géron




```
# 1. Create a model (specified to your problem)
model = tf.keras.Sequential([
  tf.keras.Input(shape=(224, 224, 3)),
  tf.keras.layers.Dense(100, activation="relu"),
  tf.keras.layers.Dense(3, activation="softmax")
])

# 2. Compile the model
model.compile(loss=tf.keras.losses.CategoricalCrossentropy(),
              optimizer=tf.keras.optimizers.Adam(),
              metrics=["accuracy"])

# 3. Fit the model
model.fit(X_train, y_train, epochs=5)

# 4. Evaluate the model
model.evaluate(X_test, y_test)
```



The ReLU (Rectified Linear Unit) activation function and the step function are both non-linear activation functions commonly used in neural networks, but they have different characteristics and purposes.

2.1. Activation Functions

Activation functions play a crucial role in artificial neural networks by introducing non-linearity to the network. They are applied to the outputs of neurons to determine whether they should be activated (i.e., "fire" or transmit a signal) or remain inactive (i.e., "not fire" or block the signal). Activation functions are essential for enabling neural networks to learn complex patterns and relationships in data.

Here are the main functions of activation functions in neural networks:

1. **Introduce Non-Linearity**: Without activation functions, the neural network would behave as a linear model, no matter how many layers it has. By applying non-linear activation functions to the outputs of neurons, the network can learn non-linear relationships in the data, making it capable of handling complex patterns and making more accurate predictions.

2. **Enable Complex Representations**: Activation functions allow neural networks to model complex input-output mappings by transforming the neuron's input to a more expressive output. This enables the network to learn and represent intricate relationships in the data.
3. **Control Neuron Activation**: The activation function determines the neuron's output based on its input. It controls whether the neuron should be activated (output a non-zero value) or remain inactive (output zero). This is essential for passing signals through the network selectively.
4. **Introduce Sparsity**: Some activation functions, like ReLU (Rectified Linear Unit), introduce sparsity by setting negative values to zero. This sparsity can help reduce computation and memory requirements during training and inference.
5. **Handle Vanishing and Exploding Gradients**: Activation functions can help mitigate the vanishing and exploding gradient problems that can occur during training. By using well-behaved activation functions, gradients can be better controlled, leading to more stable and efficient training.
6. **Normalization**: Activation functions can help normalize the output of neurons, ensuring that the activations are within a certain range and avoiding issues like exploding values.

Common activation functions include *ReLU* (Rectified Linear Unit), *sigmoid*, *tanh* (hyperbolic tangent), and *softmax*. Each activation function has its own characteristics and is suitable for specific scenarios.

In summary, activation functions introduce non-linearity, control neuron activations, enable complex representations, and help tackle gradient-related issues in neural networks. By using appropriate activation functions, neural networks can effectively learn and approximate complex patterns and relationships in data, leading to better performance in various machine learning tasks.

Some Activation functions:

1. **ReLU (Rectified Linear Unit) Activation Function**:

- Advantages: ReLU is computationally efficient and helps alleviate the vanishing gradient problem, making it popular in deep learning. It introduces sparsity by setting negative values to zero, allowing the network to learn faster and be less likely to get stuck during training.

- Disadvantages: ReLU can suffer from the "dying ReLU" problem, where neurons can become inactive (output zero) and fail to update during training. This can happen if a large gradient flows through a ReLU neuron, causing the weights to update in such a way that the neuron is always inactive.

2. **Step Function**:

- Range: The output is binary, either 0 or 1.

- Advantages: The step function is simple and binary, making it useful for some specific applications where binary outputs are required.

- Disadvantages: The step function is not differentiable at the point x = 0, which creates challenges for gradient-based optimization algorithms used in neural networks. As a result, the step function is not commonly used as an activation function in modern neural networks.

In summary, the ReLU activation function is a non-linear function that introduces sparsity and helps with the vanishing gradient problem in deep learning. It has become a standard choice for many neural network architectures. On the other hand, the step function is a simple binary function that is not differentiable and is less commonly used as an activation function in modern neural networks due to its limitations in gradient-based optimization.

2.2. Learning rates

The **learning rate** is a hyperparameter in training neural networks that controls the step size at which the optimizer updates the model's weights during the learning process. A low learning rate means that the weights are updated gradually, taking small steps, while a high learning rate means that the weights are updated more significantly, taking larger steps.

The choice of learning rate is crucial because it can significantly impact the training process and the performance of the model. Here's why increasing the learning rate may sometimes lead to better performance:

1. **Faster Convergence**: With a higher learning rate, the model's weights are updated more rapidly, which can lead to faster convergence during training. If the learning rate is too low, the training process might be slow, and the model might require more epochs to converge to a good solution.

2. **Escaping Local Minima**: In certain cases, a higher learning rate can help the model escape local minima and find better solutions. When the learning rate is low, the model might get stuck in a local minimum and fail to reach the global minimum (the optimal solution). A higher learning rate allows the model to explore more solutions and potentially find better optima.

3. **Dealing with Plateaus**: During training, the loss landscape may contain plateaus (flat regions). A low learning rate can slow down the learning process significantly in these areas. A higher learning rate can help the model overcome such plateaus more effectively.

However, increasing the learning rate significantly can have *drawbacks* as well:

1. **Overshooting**: If the learning rate is too high, the optimizer might overshoot the optimal weights and fail to converge to a good solution. The weights might oscillate and diverge, preventing the model from learning effectively.

2. **Instability**: A high learning rate can lead to unstable training, where the loss may fluctuate wildly, making it difficult for the model to converge.

3. **Skipping the Optimal Solution**: Extremely high learning rates can lead to the model missing the optimal solution altogether, as the weight updates may be too drastic.

Finding the right learning rate is often a matter of experimentation and fine-tuning. Techniques like learning rate scheduling and adaptive learning rate methods (e.g., Adam, RMSprop) can be used to automatically adjust the learning rate during training, helping strike a balance between fast convergence and stable learning.

In practice, it's common to start with a relatively small learning rate and gradually increase it if the model is not converging or is converging too slowly. However, setting the learning rate too high is generally discouraged, as it can lead to unstable training and poor results.

*How do we find the **Optimal Learning Rate**?* For this purpose we have three options:

1. **Using the predefined learning rate by tensorflow:** tensorflow generally already has as predefined a learning rate that is the most useful for most problems, but again, this can vary in certain case, for those cases we use the following methods.

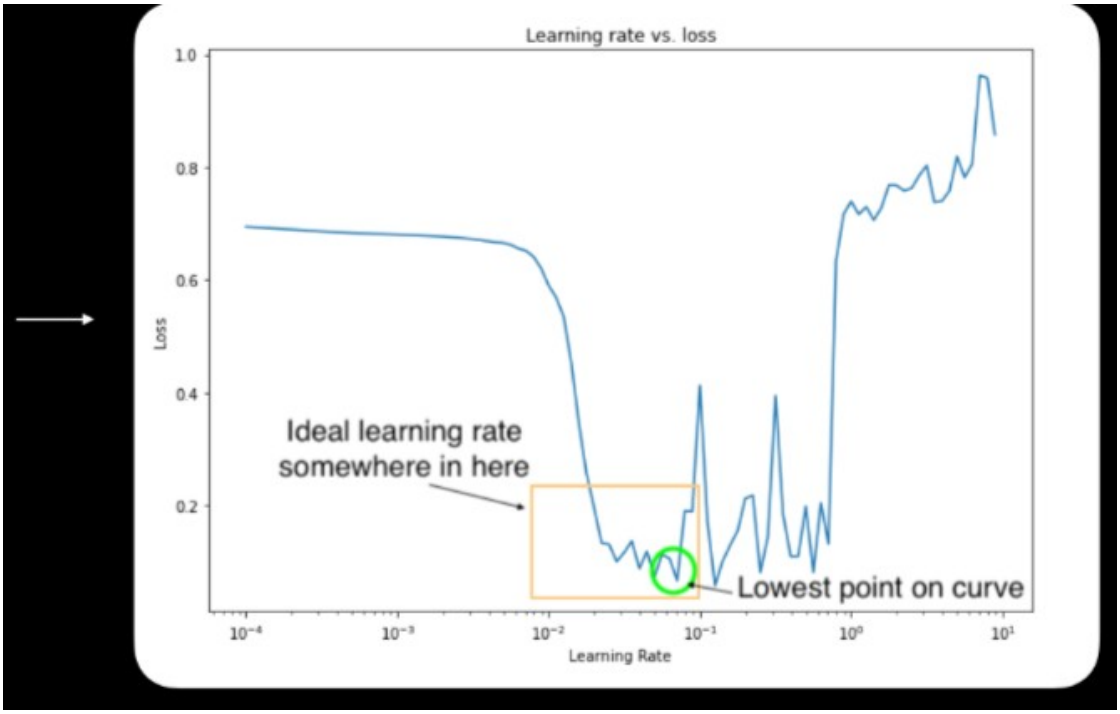
2. **Calculating the ideal learning rate**: The first thing we do is to create a *learning rate scheduler callback*, this is a callback in deep learning frameworks that dynamically adjusts the learning rate during the training process. Here's how a learning rate scheduler works:

    - *Learning Rate Initialization*: Before training the model, the learning rate scheduler starts with an initial learning rate. This value is usually set based on some intuition or common practice.
    - *Learning Rate Update*: During the training process, the learning rate scheduler monitors the progress of the training and, at predefined intervals (e.g., after each epoch or a certain number of iterations), it updates the learning rate based on a specific rule or schedule.
    - *Dynamic Learning Rate*: By adjusting the learning rate dynamically, the scheduler helps the optimization process find an optimal balance between fast convergence and avoiding overshooting or divergence.

    Take this code as example (See next page):

```
# Model code
...

# Create a learning rate scheduler callback
# (traverse set of learning rate values from 1e-4, increasing by 10**(epoch/20) every epoch)
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-4 * 10**(epoch/20))

# Fit the model (passing the lr_scheduler callback)
history = model.fit(X_train,
                    y_train,
                    epochs=100,
                    callbacks=[lr_scheduler])

# Plot the learning rate versus the loss
lrs = 1e-4 * (10 ** (np.arange(100)/20))
plt.figure(figsize=(10, 7))
plt.semilogx(lrs, history.history["loss"])
plt.xlabel("Learning Rate")
plt.ylabel("Loss")
plt.title("Learning rate vs. loss");
```

The scheduler function adjusts the learning rate for each epoch dynamically. In this example, the learning rate starts at `1e-4` (0.0001) and increases by a factor of `10^(epoch/20)` every epoch.



The ideal learning rate at the start of model training is somewhere just before the loss curve bottoms out (a value where the loss is still decreasing).

3. **Using other typical learning rate values:** We can also use the most typical learning rate values, these will work well for the majority of the problems that we might encounter.

```
[56] # Example of other typical learning rates values:
     10**0, 10**-1, 10**-2, 10**-3, 1e-4

     (1, 0.1, 0.01, 0.001, 0.0001)
```

2.3. Classification evaluation Methods

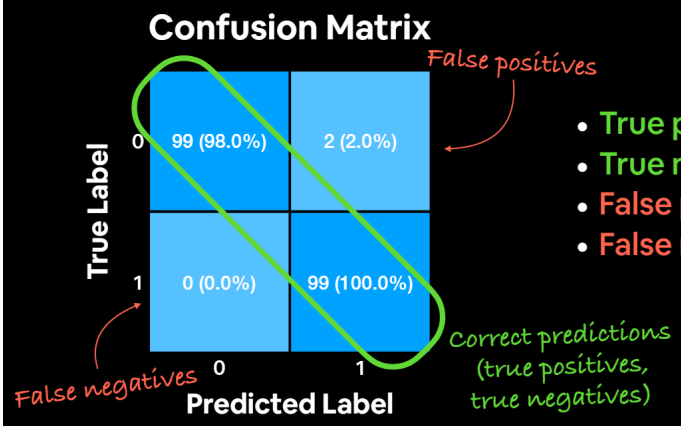
(some common)

Classification evaluation methods

Key: **tp** = True Positive, **tn** = True Negative, **fp** = False Positive, **fn** = False Negative

Metric Name	Metric Forumla	Code	When to use
Accuracy	**Accuracy** $= \frac{tp + tn}{tp + tn + fp + fn}$	`tf.keras.metrics.Accuracy()` or `sklearn.metrics.accuracy_score()`	Default metric for classification problems. Not the best for imbalanced classes.
Precision	**Precision** $= \frac{tp}{tp + fp}$	`tf.keras.metrics.Precision()` or `sklearn.metrics.precision_score()`	Higher precision leads to less false positives.
Recall	**Recall** $= \frac{tp}{tp + fn}$	`tf.keras.metrics.Recall()` or `sklearn.metrics.recall_score()`	Higher recall leads to less false negatives.
F1-score	**F1-score** $= 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$	`sklearn.metrics.f1_score()`	Combination of precision and recall, usually a good overall metric for a classification model.
Confusion matrix	NA	`Custom function` or `sklearn.metrics.confusion_matrix()`	When comparing predictions to truth labels to see where model gets confused. Can be hard to use with large numbers of classes.

Anatomy of a confusion matrix



- True positive = model predicts 1 when truth is 1
- True negative = model predicts 0 when truth is 0
- False positive = model predicts 1 when truth is 0
- False negative = model predicts 0 when truth is 1

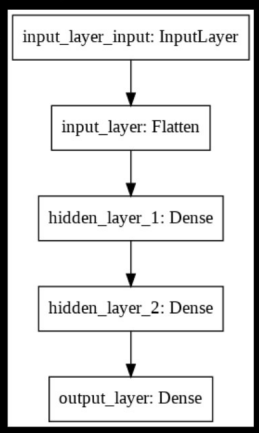
Inputs and outputs

(layer by layer)

```
[[116, 78, 15],
[117, 43, 96],
[125, 87, 23],
…,
```

Takes inputs →

```
import tensorflow as tf

# Create the model
model = tf.keras.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28), name="input_layer"),
  tf.keras.layers.Dense(4, activation="relu", name="hidden_layer_1"),
  tf.keras.layers.Dense(4, activation="relu", name="hidden_layer_2"),
  tf.keras.layers.Dense(10, activation="softmax", name="output_layer")
], name="10_class_classification_model")

# Plot the model
tf.keras.utils.plot_model(model)
```



Outputs to

Outputs to

Outputs to

Outputs to

Outputs to

```
[[0.983, 0.004, 0.013],
[0.110, 0.889, 0.001],
[0.023, 0.027, 0.985],
…,
```

3. Computer Vision and Convolutional Neural Networks

Architecture of a CNN:

Hyperparameter/Layer type	What does it do?	Typical values
Input image(s)	Target images you'd like to discover patterns in	Whatever you can take a photo (or video) of
Input layer	Takes in target images and preprocesses them for further layers	`input_shape = [batch_size, image_height, image_width, color_channels]`
Convolution layer	Extracts/learns the most important features from target imagaes	Multiple, can create with `tf.keras.layers.ConvXD` (X can be multiple values)
Hidden activation	Adds non-linearity to learned features (non-straight lines)	Usually ReLU (`tf.keras.activations.relu`)
Pooling layer	Reduces the dimensionality of learned image features	Average (`tf.keras.layers.AvgPool2D`) or Max (`tf.keras.layers.MaxPool2D`)
Fully connected layer	Further refines learned features from convolution layers	`tf.keras.layers.Dense`
Output layer	Takes learned features and outputs them in shape of target labels	`output_shape = [number_of_classes]` (e.g. 3 for pizza, steak or sushi)
Output activation	Adds non-linearities to output layer	`tf.keras.activations.sigmoid` (binary classification) or `tf.keras.activations.softmax`

```
# 1. Create a CNN model (same as Tiny VGG - https://poloclub.github.io/cnn-explainer/)
cnn_model = tf.keras.models.Sequential([
  tf.keras.layers.Conv2D(filters=10,
                         kernel_size=3, # can also be (3, 3)
                         activation="relu",
                         input_shape=(224, 224, 3)), # specify input shape (height, width, colour channels)
  tf.keras.layers.Conv2D(10, 3, activation="relu"),
  tf.keras.layers.MaxPool2D(pool_size=2, # pool_size can also be (2, 2)
                            padding="valid"), # padding can also be 'same'
  tf.keras.layers.Conv2D(10, 3, activation="relu"),
  tf.keras.layers.Conv2D(10, 3, activation="relu"), # activation='relu' == tf.keras.layers.Activations(tf.nn.relu)
  tf.keras.layers.MaxPool2D(2),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(1, activation="sigmoid") # binary activation output
])

# 2. Compile the model
cnn_model.compile(loss="binary_crossentropy",
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=["accuracy"])

# 3. Fit the model
history = cnn_model.fit(train_data, epochs=5)
```

(*Note:* there are almost an unlimited amount of ways we could stack together a convolutional neural network, this is just one).

Both traditional neural networks and convolutional neural networks (CNNs) can be used for image classification tasks, including the Fashion MNIST dataset. The choice between the two depends on various factors, including the complexity of the dataset, the architectural requirements, and the desired performance.

For the Fashion MNIST dataset, both neural networks and CNNs can yield good results. However, CNNs are generally more suited for image-related tasks due to their ability to automatically learn hierarchical features from the data, particularly spatial and translation-invariant features. CNNs are specifically designed to handle grid-like data structures such as images, and they leverage operations like convolution and pooling to capture local patterns and structures.

Here's why a CNN might be preferred for image classification tasks like Fashion MNIST:

1. **Spatial Hierarchies**: Images often have spatial hierarchies of features, where lower-level features (e.g., edges, textures) combine to form higher-level features (e.g., shapes, objects). CNNs are particularly effective in capturing such hierarchies through their convolutional layers.
2. **Translation Invariance**: CNNs are capable of learning translation-invariant features. This means that if a certain feature (e.g., a specific pattern) is relevant in one part of the image, the CNN can detect the same feature in a different part of the image, even if it's shifted.
3. **Parameter Efficiency**: CNNs use weight sharing in their convolutional layers, which reduces the number of parameters compared to fully connected layers in traditional neural networks. This makes CNNs more efficient for image data, where local patterns repeat across the image.
4. **Dimensionality Reduction**: Pooling layers in CNNs perform downsampling, reducing the spatial dimensions of the data. This can help in focusing on the most important features while reducing the computational load.
5. **Feature Learning**: CNNs automatically learn relevant features from the data, reducing the need for manual feature engineering. This is particularly advantageous for complex image datasets like Fashion MNIST.

That being said, for simpler datasets like Fashion MNIST, a traditional neural network could still achieve reasonable accuracy, especially if the image features are not highly complex. The decision between using a traditional neural network and a CNN often depends on the balance between model complexity and performance. For more complex and larger image datasets, CNNs tend to shine due to their ability to capture intricate patterns and relationships.

3.1. Scaling (normalizing) images

Normalizing images is an important preprocessing step in training convolutional neural networks (CNNs) for several reasons:

1. **Stability and Convergence**: Normalizing the pixel values of images to a similar range (usually [0, 1] or [-1, 1]) helps to ensure that the input data has a consistent scale. This can lead to faster convergence during training, as the optimization algorithms can work more effectively with inputs that are within a similar numerical range.
2. **Gradient Descent**: Gradient descent optimization algorithms (like SGD, Adam, etc.) can perform better when the input data is centered around zero and has a small variance. This can help prevent the optimization process from getting stuck in local minima or converging too slowly.
3. **Regularization**: Normalization can act as a form of regularization by preventing large pixel values from dominating the learning process. It can help in controlling the magnitude of weight updates during training.
4. **Avoiding Numerical Instabilities**: Large pixel values can lead to numerical instabilities in the intermediate computations of the network, particularly in deep architectures where large values could result in overflow or underflow issues.

5. **Weight Initialization**: Normalizing the inputs can make it easier to initialize the network's weights in a way that prevents vanishing or exploding gradients.
6. **Generalization**: Normalization can improve the model's generalization performance by making it less sensitive to variations in input intensity levels, lighting conditions, and color variations.
7. **Different Activation Functions**: Different activation functions (e.g., ReLU, sigmoid, tanh) respond differently to input values. Normalizing inputs helps to ensure that activation functions are operating in their more linear and efficient regions, which can lead to improved learning.

It's important to note that different normalization strategies can be employed depending on the use case and the nature of the data.

In Convolutional Neural Networks (CNNs), the batch size is an important hyperparameter that determines the number of training examples used in each iteration of gradient descent during training. It has several advantages and purposes in the context of CNNs and deep learning in general:

1. **Memory Efficiency**: CNNs often deal with large datasets and complex models. Loading the entire dataset into memory can be impractical or even impossible due to memory limitations. By using batch processing, you can work with a smaller subset of the data at a time, allowing your model to fit within the available memory.
2. **Faster Convergence**: Training a deep neural network can be computationally intensive. Using smaller batch sizes allows you to update the model's parameters more frequently, which can lead to faster convergence since parameter updates are based on more recent data.
3. **Generalization**: Training on larger batch sizes can sometimes result in overfitting, where the model memorizes the training data rather than learning meaningful patterns. Smaller batch sizes can act as a regularizer, preventing the model from becoming too specialized to the training data.

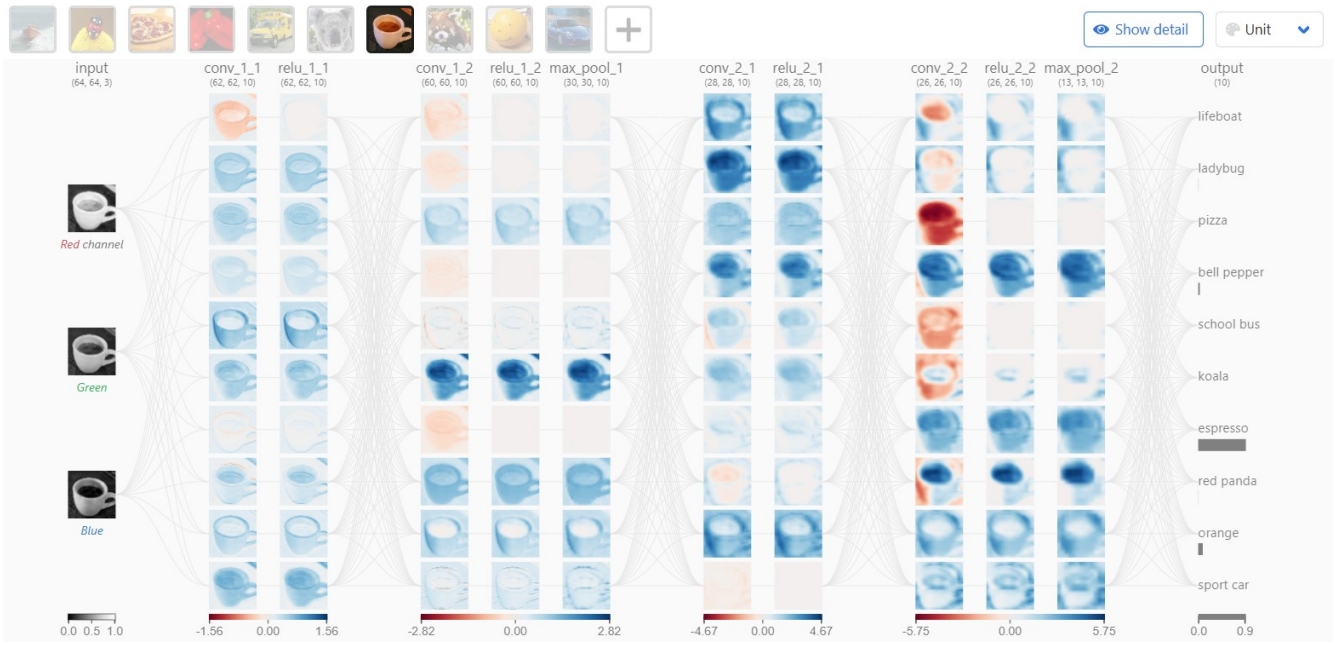
4. **Parallelization**: Many deep learning frameworks and hardware accelerators support parallelism, allowing multiple computations to be performed simultaneously. Smaller batch sizes can take advantage of this parallelism, speeding up training on compatible hardware.
5. **Stochastic Gradient Descent (SGD)**: Batch processing is a key aspect of SGD, which is the optimization algorithm commonly used to update the model's parameters. The stochastic nature of using mini-batches introduces a level of randomness that can help the optimization process escape local minima.
6. **Noise Reduction**: Using smaller batch sizes introduces more noise to the optimization process due to the randomness of each batch. This can help the model generalize better, as it learns to adapt to a variety of data patterns.

However, there are also some considerations to keep in mind when choosing a batch size:

- Smaller batch sizes can lead to noisy gradient estimates, potentially causing training instability.
- Larger batch sizes can lead to more accurate gradient estimates but require more memory and computational resources.
- The ideal batch size can vary depending on the dataset, model complexity, and hardware available.

3.2. An in-depth explanation of CNNs

CNN Explainer (poloclub.github.io/cnn-explainer/)

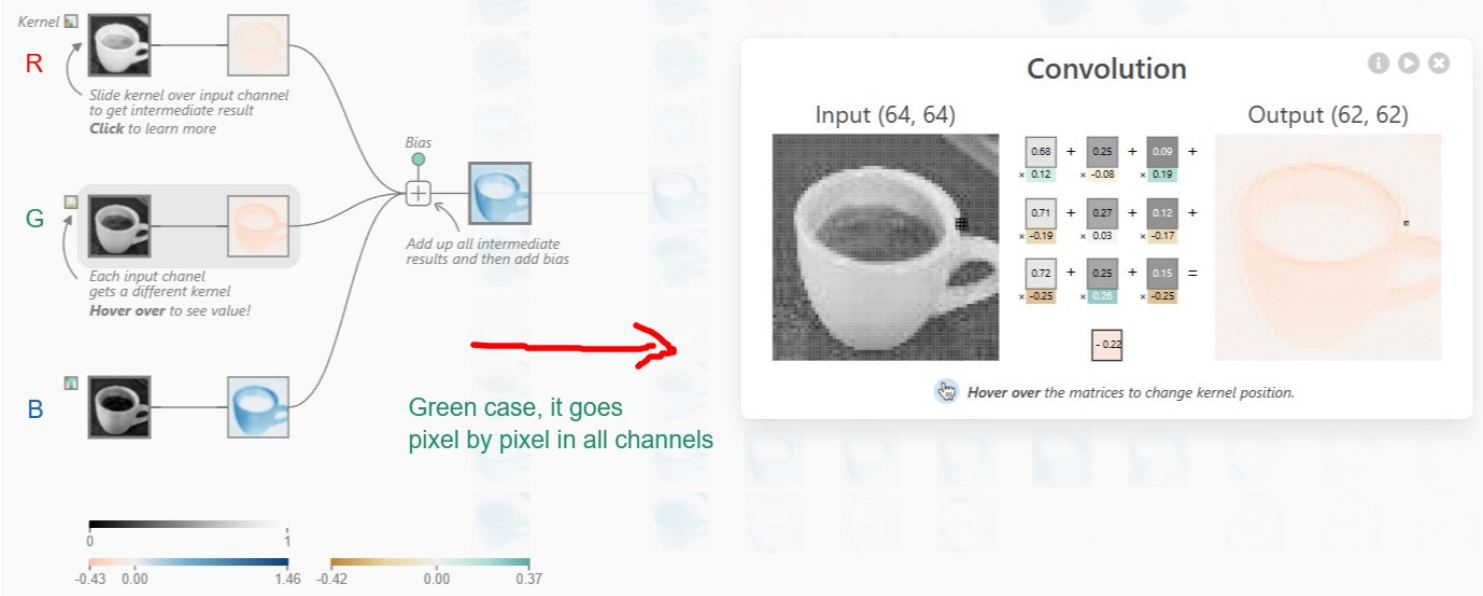


Key points:

1. **Kernel weights and biases:** while unique to each neuron, are tuned during the training phase, and allow the classifier to adapt to the problem and dataset provided.

2. The convolutional layers are the foundation of CNN, as they contain the learned kernels (weights), which extract features that distinguish different images from one another—this is what we want for classification! As you interact with the convolutional layer, you will notice links between the previous layers and the convolutional layers. **Each link represents a unique kernel,** which is used for the convolution operation to produce the current convolutional neuron's output or activation map.

The convolutional neuron performs an elementwise dot product with a unique kernel and the output of the previous layer's corresponding neuron. This will yield as many intermediate results as there are unique kernels. **The convolutional neuron is the result of all of the intermediate results summed together with the learned bias.**



The size of these kernels is a hyper-parameter specified by the designers of the network architecture. In order to produce the output of the convolutional neuron (activation map), we must perform an elementwise dot product with the output of the previous layer and the unique kernel learned by the network.

In TinyVGG, the dot product operation uses a stride of 1, which means that the kernel is shifted over 1 pixel per dot product, but this is a hyperparameter that the network architecture designer can adjust to better fit their dataset. We must do this for all 3 kernels, which will yield 3 intermediate results.

Why do we do this?

The use of the dot product operation in convolutional layers of a neural network is a fundamental concept in convolutional neural networks (CNNs), and it serves several important purposes:

1. **Feature Detection**: The dot product operation between a kernel (also known as a filter) and a portion of the input data results in a value that represents the similarity between the kernel and that portion of the data. By sliding the kernel over the entire input data, the network learns to detect different features and patterns present in the data.
2. **Local Receptive Fields**: The dot product operation creates a local receptive field for each neuron in the convolutional layer. Each neuron's kernel focuses on a small portion of the input data. This enables the network to capture local patterns, such as edges, corners, and textures, which are building blocks for more complex features.
3. **Weight Sharing**: The use of the same kernel across different regions of the input data allows the network to share learned features. This makes the model more efficient and effective in capturing hierarchical features in different parts of the input.
4. **Translation Invariance**: The convolution operation is translation invariant, meaning it can detect patterns regardless of their location in the input. This property is crucial for tasks such as image recognition, where the position of a feature may vary within an image.
5. **Reduced Parameter Count**: By using a small kernel and sliding it over the input data, the network learns a relatively small number of parameters compared to a fully connected layer. This reduces the risk of overfitting and makes the model more efficient.

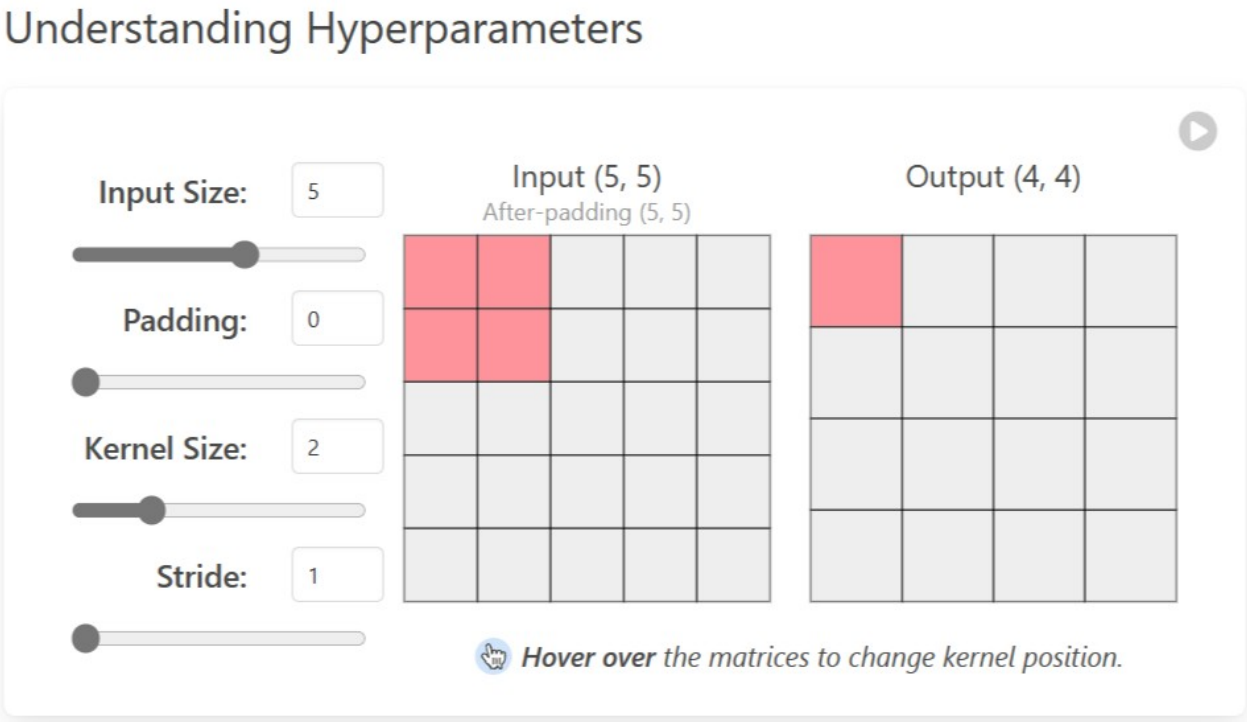
Convolution Operation: The kernel slides (or convolves) across the input image, and at each position, it performs an element-wise multiplication (dot product) between its weights and the values of the input pixels it overlaps with.

1. **Summation**: After the element-wise multiplications are performed for a specific position of the kernel, the results are summed up. This sum represents the contribution of the kernel at that position to the output feature map.
2. **Adding Bias**: Once the sum has been computed, the bias term associated with the kernel is added to this sum. The bias term is a learnable parameter for each kernel and is used to introduce an offset to the computed sum. This allows the neural network to model linear patterns that might not be captured by the weights of the kernel.
3. **Activation Function**: After adding the bias, the result typically goes through an activation function, such as ReLU (Rectified Linear Unit), to introduce non-linearity into the network. This activation function helps the network learn complex patterns and relationships in the data.
4. **Result**: The final result is the output value at that specific position of the output feature map.

This process is repeated for each position of the kernel as it slides across the input image, resulting in the generation of the output feature map. Each kernel in a convolutional layer has its own set of weights (including a bias term), which allows the network to learn different patterns and features at different locations within the input data.

3. Then, an elementwise sum is performed containing all 3 intermediate results along with the bias the network has learned. After this, the resulting 2-dimensional tensor will be the activation map viewable on the interface above for the topmost neuron in the first convolutional layer. This same operation must be applied to produce each neuron's activation map.

4. With some simple math, we are able to deduce that there are 3 x 10 = 30 unique kernels, each of size 3x3, applied in the first convolutional layer. The connectivity between the convolutional layer and the previous layer is a design decision when building a network architecture, which will affect the number of kernels per convolutional layer.



- ***Padding:*** **is often necessary when the kernel extends beyond the activation map. Padding conserves data at the borders of activation maps, which leads to better performance, and it can help preserve the input's spatial size, which allows an architecture designer to build depper, higher performing networks.**

There exist many padding techniques, but the most commonly used approach is zero-padding because of its performance, simplicity, and computational efficiency. The technique involves adding zeros symmetrically around the edges of an input. This approach is adopted by many high-performing CNNs such as AlexNet.

- **Kernel size**: **often also referred to as filter size, refers to the dimensions of the sliding window over the input.** Choosing this hyperparameter has a massive impact on the image classification task. For example, **small kernel sizes are able to extract a much larger amount of information containing highly local features from the input.**

  As you can see on the visualization above, *a smaller kernel size also leads to a smaller reduction in layer dimensions, which allows for a deeper architecture*. Conversely, *a large kernel size extracts less information, which leads to a faster reduction in layer dimensions, often leading to worse performance*.

  Large kernels are better suited to extract features that are larger. At the end of the day, choosing an appropriate kernel size will be dependent on your task and dataset, but generally, smaller kernel sizes lead to better performance for the image classification task because an architecture designer is able to stack more and more layers together to learn more and more complex features!

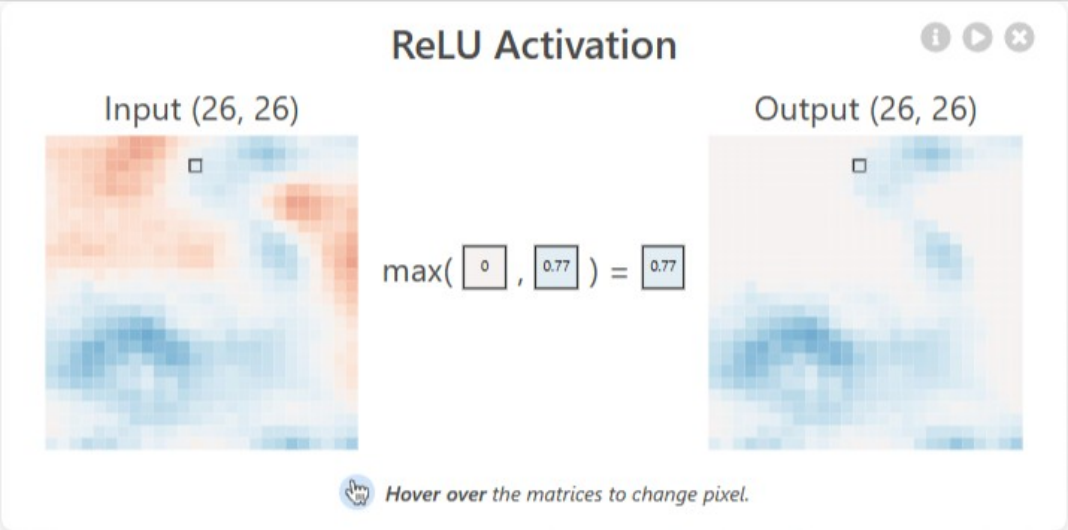
- ***Stride:*** **indicates how many pixels the kernel should be shifted over at a time.** For example, as described in the convolutional layer example above, Tiny VGG uses a stride of 1 for its convolutional layers, which means that the dot product is performed on a 3x3 window of the input to yield an output value, then is shifted to the right by one pixel for every subsequent operation.

The impact stride has on a CNN is similar to kernel size. As stride is decreased, more features are learned because more data is extracted, which also leads to larger output layers. On the contrary, *as stride is increased, this leads to more limited feature extraction and smaller output layer dimensions*.

One responsibility of the architecture designer is to ensure that the kernel slides across the input symmetrically when implementing a CNN. Use the hyperparameter visualization above to alter stride on various input/kernel dimensions to understand this constraint!

5. Some activation functions notes:

Relu:



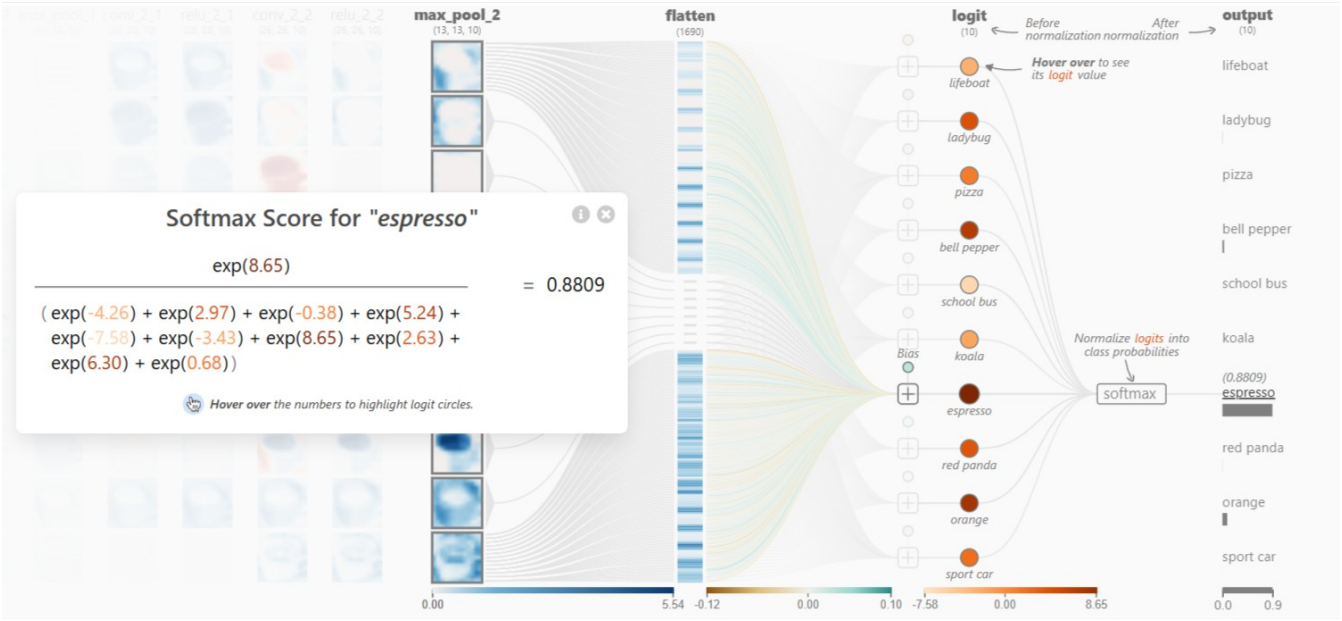
All orange would output 0 in this case.

Softmax:

A softmax operation serves a key purpose: making sure the CNN outputs sum to 1. Because of this, softmax operations are useful to scale model outputs into

probabilities. Clicking on the last layer reveals the softmax operation in the network. Notice how the logits after flatten aren't scaled between zero to one. For a visual indication of the impact of each logit (unscaled scalar value), they are encoded using a light orange → dark orange color scale. After passing through the softmax function, each class now corresponds to an appropriate probability!

You might be thinking *what the difference between standard normalization and softmax is*—after all, both rescale the logits between 0 and 1. Remember that backpropagation is a key aspect of training neural networks—we want the correct answer to have the largest "signal." By using softmax, we are effectively "approximating" argmax while gaining differentiability. Rescaling doesn't weigh the max significantly higher than other logits, whereas softmax does. Simply put, softmax is a "softer" argmax—see what we did there?



6. **Pooling:** There are many types of pooling layers in different CNN architectures, but they all have the purpose of gradually decreasing the spatial extent of the network, which reduces the parameters and overall computation of the network. The type of pooling used in the Tiny VGG architecture above is Max-Pooling.

The Max-Pooling operation requires selecting a kernel size and a stride length during architecture design. **Once selected, the operation slides the kernel with the specified stride over the input while only selecting the largest value at each kernel slice from the input to yield a value for the output.** This process can be viewed by clicking a pooling neuron in the network above.

In the Tiny VGG architecture above, the pooling layers use a 2x2 kernel and a stride of 2. This operation with these specifications results in the discarding of 75% of activations. By discarding so many values, Tiny VGG is more computationally efficient and avoids overfitting.

7. **Flatten Layer**: This layer converts a three-dimensional layer in the network into a one-dimensional vector to fit the input of a fully-connected layer for classification. For example, a 5x5x2 tensor would be converted into a vector of size 50. The previous convolutional layers of the network extracted the features from the input image, but now it is time to classify the features. We use the softmax function to classify these features, which requires a 1-dimensional input. This is why the flatten layer is necessary. This layer can be viewed by clicking any output class.

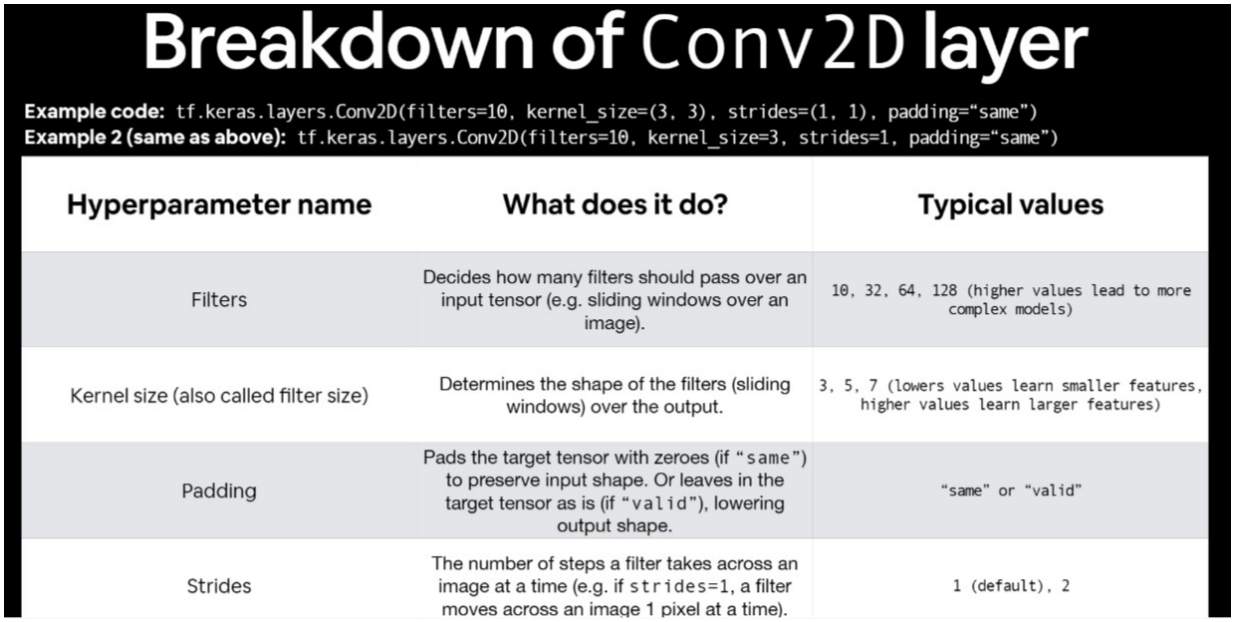
END OF KEY POINTS

3.3. Continuing notes

In my examples I use the activation functions on the same layer as each hidden unit, but we can also use them as their own separate layer, as in the example of the previous subsection.

Here's why a CNN might be preferred for image classification tasks like Fashion MNIST:

1. **Spatial Hierarchies**: Images often have spatial hierarchies of features, where lower-level features (e.g., edges, textures) combine to form higher-level features (e.g., shapes, objects). CNNs are particularly effective in capturing such hierarchies through their convolutional layers.
2. **Translation Invariance**: CNNs are capable of learning translation-invariant features. This means that if a certain feature (e.g., a specific pattern) is relevant in one part of the image, the CNN can detect the same feature in a different part of the image, even if it's shifted.
3. **Parameter Efficiency**: CNNs use weight sharing in their convolutional layers, which reduces the number of parameters compared to fully connected layers in traditional neural networks. This makes CNNs more efficient for image data, where local patterns repeat across the image.
4. **Dimensionality Reduction**: Pooling layers in CNNs perform downsampling, reducing the spatial dimensions of the data. This can help in focusing on the most important features while reducing the computational load.
5. **Feature Learning**: CNNs automatically learn relevant features from the data, reducing the need for manual feature engineering. This is particularly advantageous for complex image datasets like Fashion MNIST.



In TensorFlow, **cloning a model** refers to creating a new model that has the same architecture and initial weights as an existing model. This operation is useful in various scenarios, such as transfer learning, model ensembling, or when you want to make different modifications to the same base model without affecting the original.

3.4. Improving a model from a data perspective

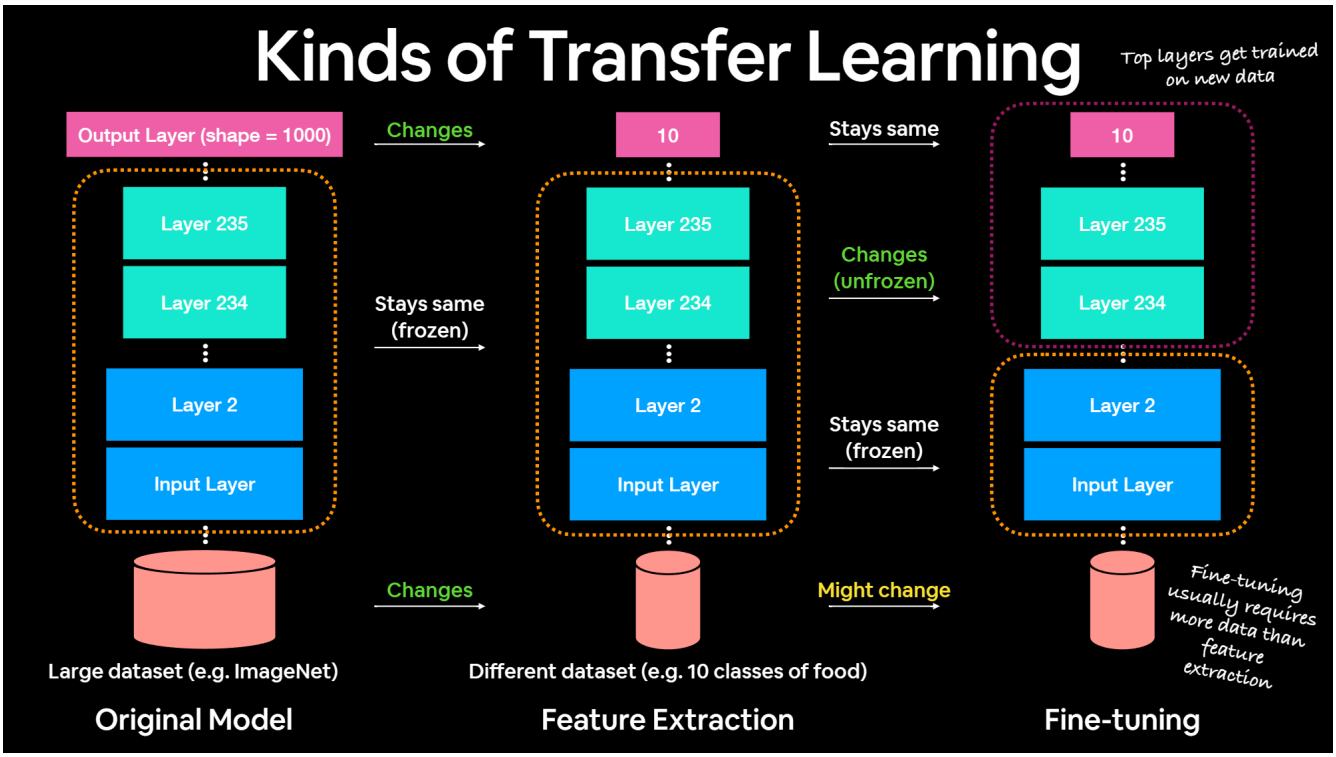
Method to improve a model (reduce overfitting)	What does it do?
More data	Gives a model more of a chance to learn patterns between samples (e.g. if a model is performing poorly on images of pizza, show it more images of pizza).
Data augmentation	Increase the diversity of your training dataset without collecting more data (e.g. take your photos of pizza and randomly rotate them 30°). Increased diversity forces a model to learn more generalisation patterns.
Better data	Not all data samples are created equally. Removing poor samples from or adding better samples to your dataset can improve your model's performance.
Use transfer learning	Take a model's pre-learned patterns from one problem and tweak them to suit your own problem. For example, take a model trained on pictures of cars to recognise pictures of trucks.

It may take longer to train our model when we **augment our data**, this is because our original data is not necessarily augmented, and that takes a little additional computing power (also, the augmented data tends to not get stored and rather is done in the fly).

Shuffling the data ensures that the order of the data samples presented to the network during training is random. Without shuffling, if the data is ordered in any particular way (e.g., all samples of one class followed by another), it could introduce bias into the learning process. Shuffling helps to eliminate this bias, making the training process more robust**.**

4. Transfer Learning

Transfer learning consists of using pre-written architectures for our models, we look at arhitectures that are proven to work well for our type of problem and we adapt them as necessary.



(Note: Pages like tensorflow hub, or paperswithcode.com give a repository of trained machine learning models ready for fine-tuning and deployable anywhere. This serves to reuse models into your own problems in the case of the former. In the case of the later, this page gives us a variety of written papers for certain kinds of problems)

4.1. Feature extraction

Feature extraction involves using a pre-trained model as a fixed feature extractor. You remove the final classification layers (head) of the pre-trained model and replace them with new layers that are specific to your target task.

- The original weights of the pre-trained model's layers are frozen, which means they are not updated during training. Only the newly added layers are trained on the new dataset.
- This approach is often used when you have a relatively small dataset for your target task. By using the feature extraction method, you benefit from the pre-trained model's ability to extract meaningful features from the data while customizing the classification layers for your specific problem.
- It's an effective technique when you have limited data, as fine-tuning the entire model on a small dataset can lead to overfitting.

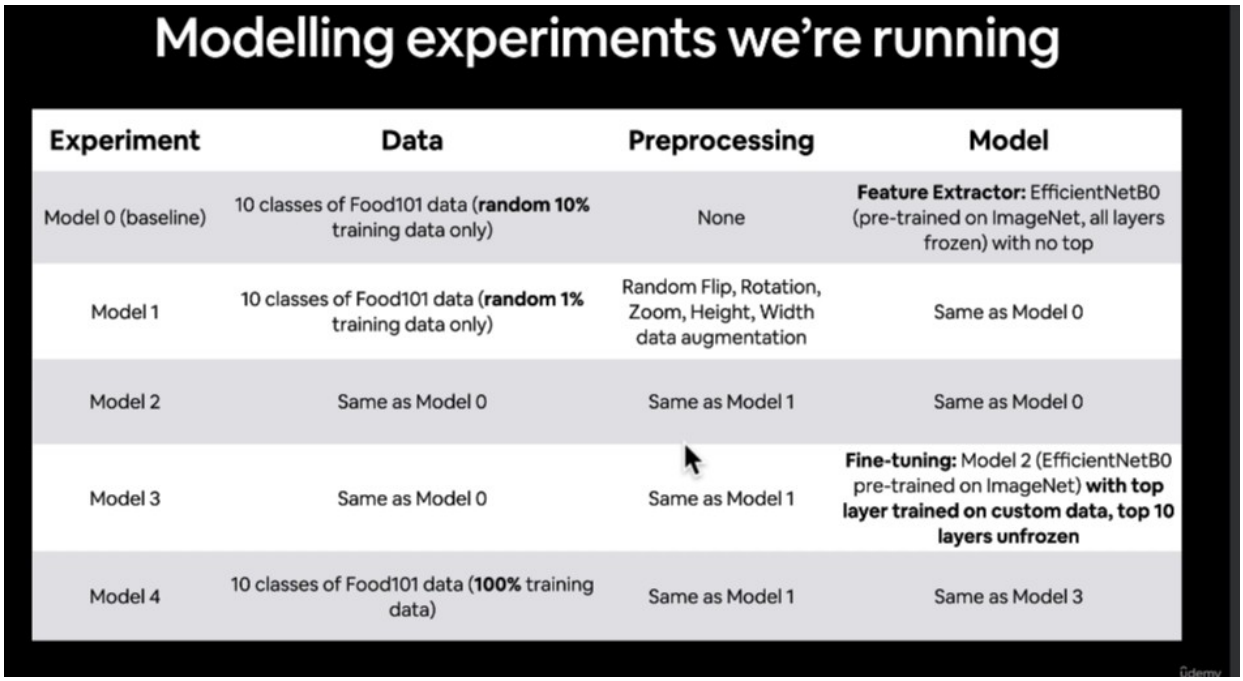
4.2. Fine-Tuning

Fine-tuning is a more extensive adaptation of the pre-trained model. Instead of keeping the entire pre-trained model frozen, you unfreeze some of the layers in the model, typically closer to the input layers, and allow them to be updated during training.

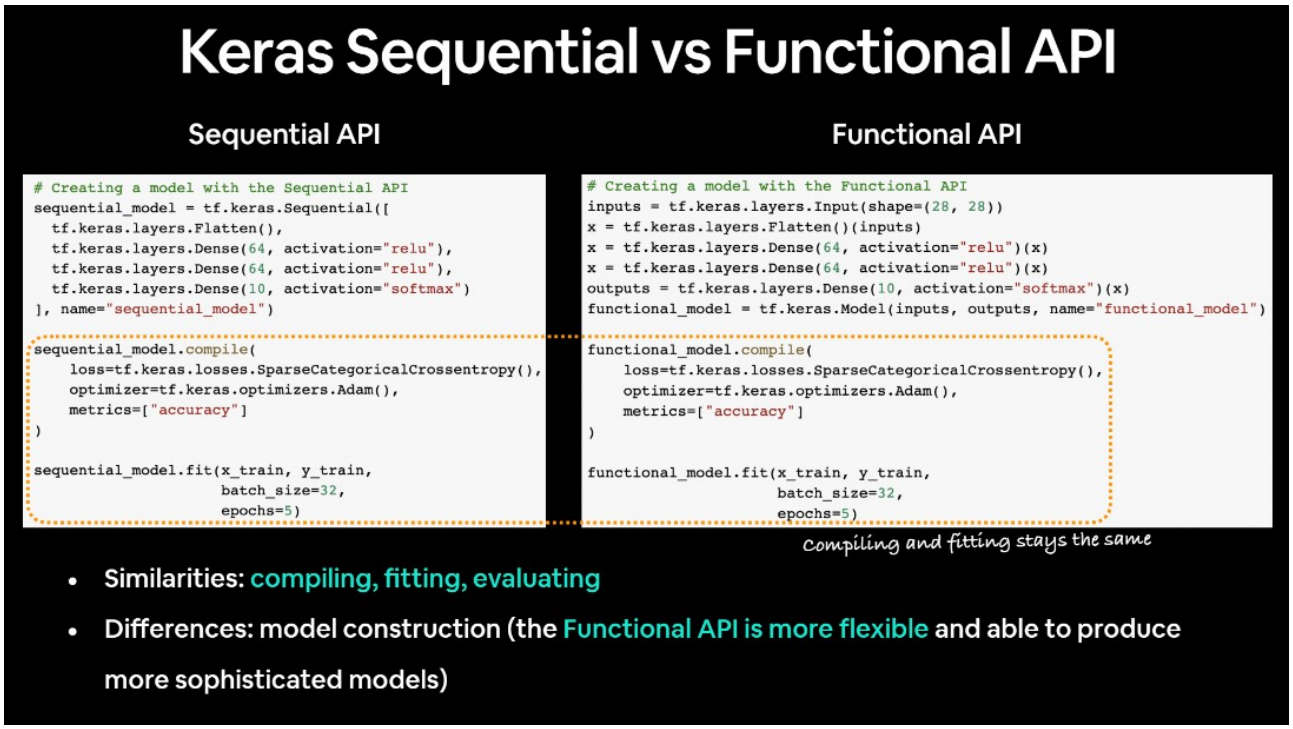
You then train the entire model, including the newly unfrozen layers, on your target task with your dataset.

Fine-tuning is usually applied when you have a larger dataset for your target task because it involves updating a larger portion of the model's parameters. The idea is

that the model can learn task-specific features while still retaining valuable knowledge from the pre-trained layers. Fine-tuning is more computationally intensive and requires a larger dataset compared to feature extraction.



In this module we will start using **the functional API:**



Sequential API:

1. Linear Stacking of Layers:

- **Usage:** The Sequential API is appropriate for linear stacks of layers, where each layer has exactly one input tensor and one output tensor.

- **Simplicity:** It is very straightforward and easy to use, making it a good choice for simple models with a single input and a single output.

2. Single Input and Single Output:

- Structure: Layers are added one by one in a linear manner. Each layer receives the output from the previous layer.

- Limitations: It is less flexible when dealing with models that have multiple inputs, multiple outputs, shared layers, or non-sequential connectivity.

Functional API:

1. Graph-Like Models:

- Usage: The Functional API is more flexible and can handle models with complex topologies, including models with multiple inputs and outputs, shared layers, and residual connections.

- Versatility: It allows the creation of directed acyclic graphs of layers, enabling the construction of more intricate models.

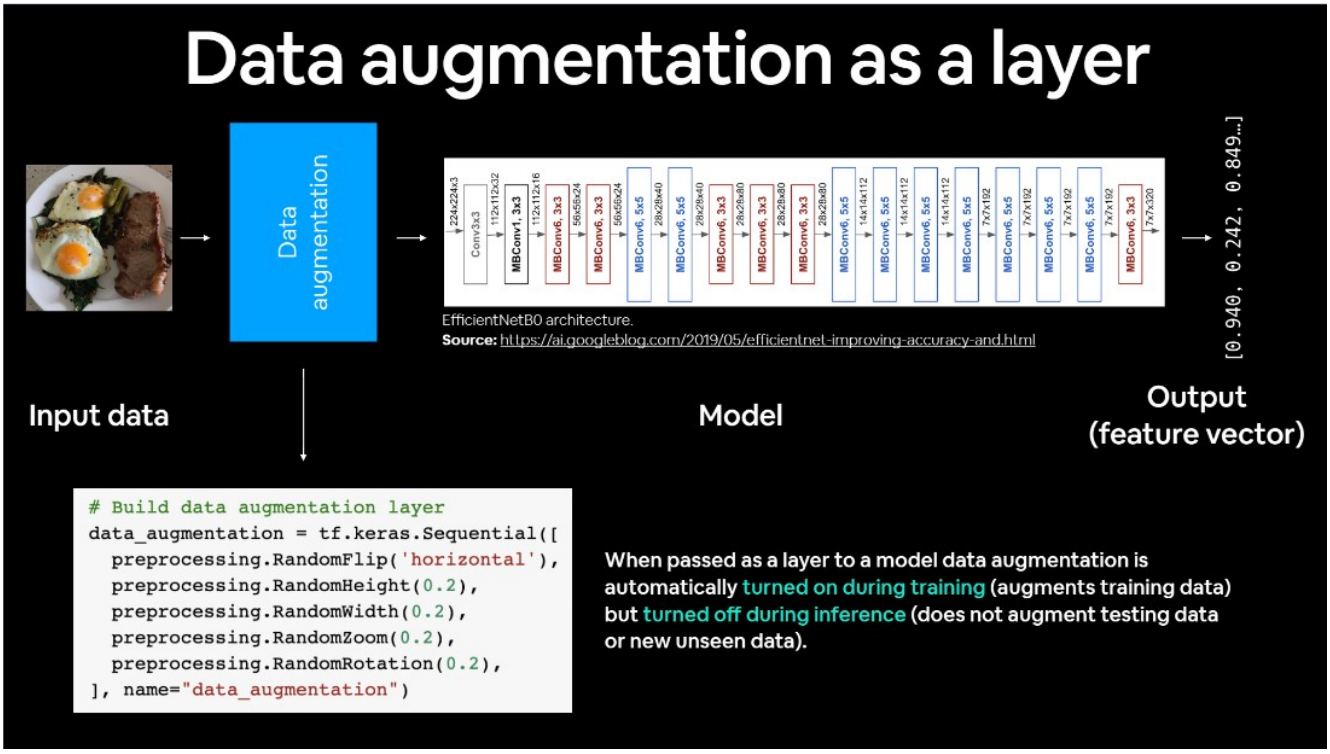
2. Multiple Inputs and Outputs:

- Structure: Layers are defined as functions that take one or more input tensors and produce one or more output tensors. These layers can then be connected to form a directed acyclic graph.

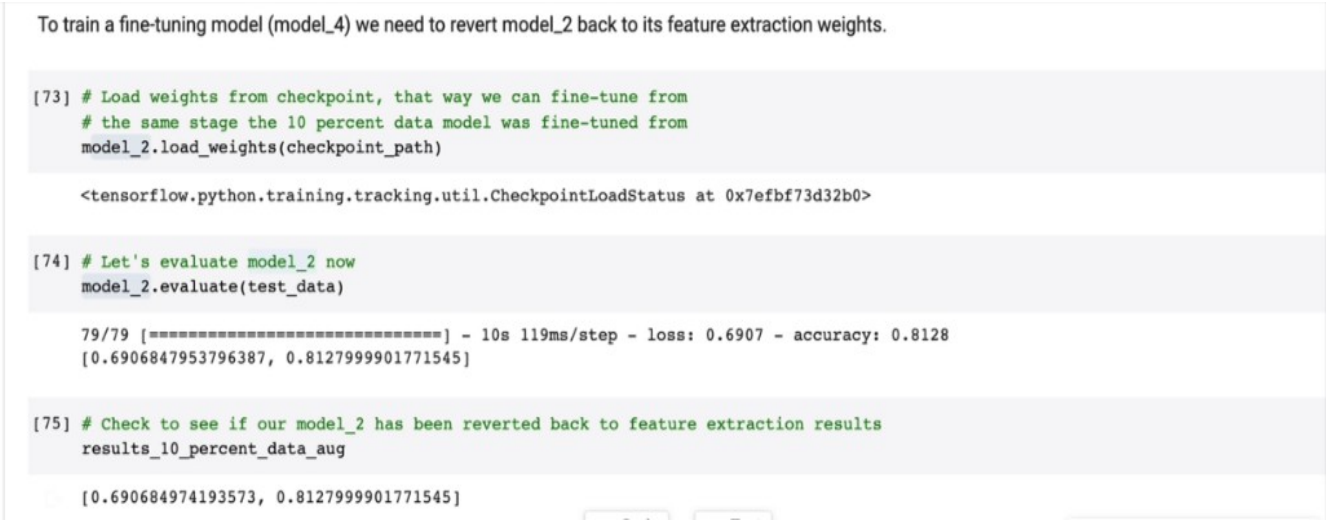
- Flexibility: It allows for the creation of models with multiple input branches and multiple output branches.

- Necessary for more complex models with multiple inputs and outputs, shared layers, or non-sequential connectivity.

- Offers greater flexibility and is essential for building advanced architectures.



Before we fine tune any layers in existing architecture, such as efficient net zero, what you want to do first, *build a feature extractor model first.* You train the weights in this output layer *and then you unfreeze some of the layers*, you keep this layered, trained on your custom data, and then you start to work backwards and unfreeze some of the underlying layers. So that's generally the workflow for fine tuning.



Why do we lower the learning rate in order to perform fine-tuning?

1. *Stability during Fine-Tuning:* Fine-tuning involves training a pre-trained model on a new dataset, often with a different distribution or characteristics than the original dataset used for pre-training. Lowering the learning rate helps stabilize the training process, preventing large updates to the model weights that might disrupt the already learned features.
2. *Avoiding Catastrophic Forgetting:* Catastrophic forgetting refers to the phenomenon where the fine-tuning process on a new dataset leads the model to forget previously learned knowledge from the pre-training phase. Lowering the learning rate helps mitigate this issue by allowing the model to adapt to the new data while retaining valuable knowledge from the pre-trained weights.
3. *Smaller Steps in Weight Space:* Fine-tuning typically involves adjusting the model's parameters to better align with the specifics of the new task or dataset. Smaller learning rates result in smaller steps in the weight space, making the fine-tuning process more controlled and preventing the model from making drastic changes.

4. *Preventing Overfitting*: Lowering the learning rate can act as a regularization technique during fine-tuning, helping to prevent overfitting to the new dataset. Smaller learning rates make the model less prone to fitting noise in the new data, especially when the amount of new data is limited.

5. *Refinement of Learned Features:* During fine-tuning, the model's lower layers, which capture general features, are often frozen to retain the knowledge learned from the original task. Lowering the learning rate allows the higher layers (closer to the output) to be more flexible and adapt to the specifics of the new task.

4.3. Scaling up

Mixed precision is the use of both 16-bit and 32-bit floating-point types in a model during training to make it run faster and use less memory. By keeping certain parts of the model in the 32-bit types for numeric stability, the model will have a lower step time and train equally as well in terms of the evaluation metrics such as accuracy. **Using this can improve performance by more than 3 times on modern GPUs, 60% on TPUs and more than 2 times on latest Intel CPUs.**

Today, most models use the float32 dtype, which takes 32 bits of memory. However, there are two lower-precision dtypes, float16 and bfloat16, each which take 16 bits of memory instead. Modern accelerators can run operations faster in the 16-bit dtypes, as they have specialized hardware to run 16-bit computations and 16-bit dtypes can be read from memory faster.

NVIDIA GPUs can run operations in float16 faster than in float32, and TPUs and supporting Intel CPUs can run operations in bfloat16 faster than float32. Therefore, these lower-precision dtypes should be used whenever possible on those devices.

However, variables and a few computations should still be in float32 for numeric reasons so that the model trains to the same quality. The Keras mixed precision API allows you to use a mix of either float16 or bfloat16 with float32, to get the performance benefits from float16/bfloat16 and the numeric stability benefits from float32.

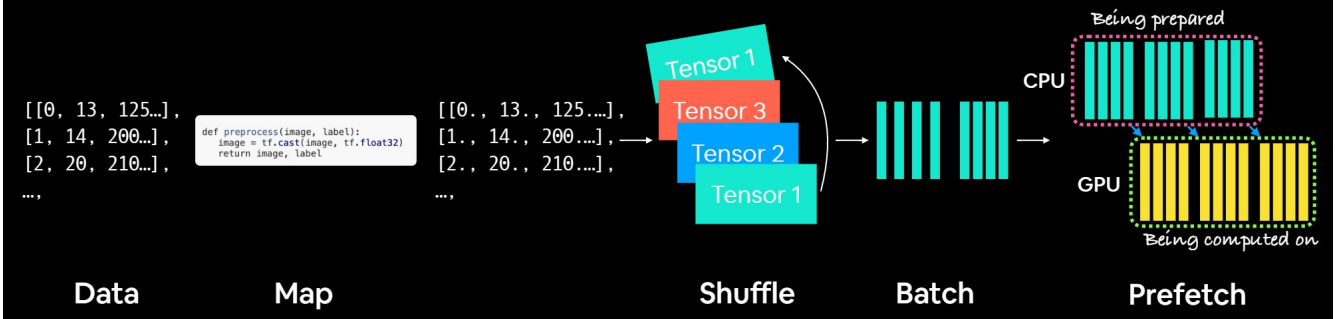
While mixed precision will run on most hardware, it will only speed up models on recent NVIDIA GPUs, Cloud TPUs and recent Intel CPUs. NVIDIA GPUs support using a mix of float16 and float32, while TPUs and Intel CPUs support a mix of bfloat16 and float32.

Among NVIDIA GPUs, those with compute capability 7.0 or higher will see the greatest performance benefit from mixed precision because they have special hardware units, called Tensor Cores, to accelerate float16 matrix multiplications and convolutions.

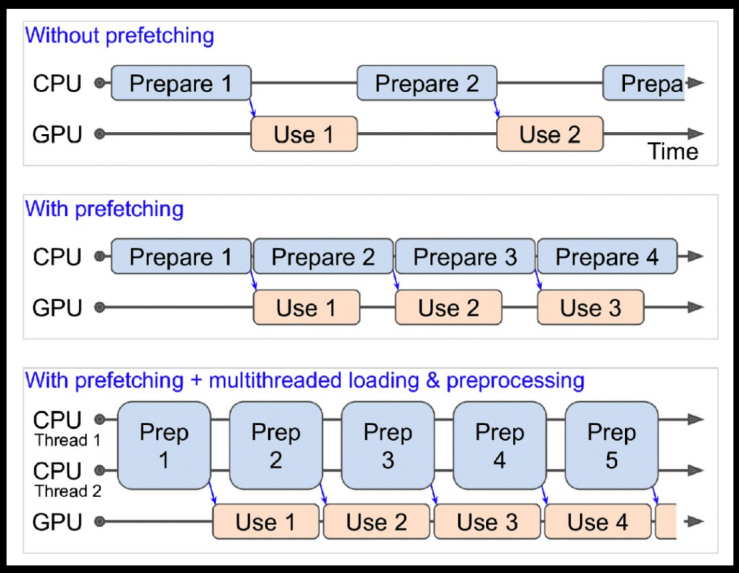
When loading a dataset from TFDS (Tensorflow datasets) you might have to do some stuff to preprocess:

Batching & preparing datasets

```
# Map preprocessing function to training data (and paralellize)
train_data = train_data.map(map_func=preprocess, num_parallel_calls=tf.data.AUTOTUNE)
# Shuffle train_data and turn it into batches and prefetch it (load it faster)
train_data = train_data.shuffle(buffer_size=1000).batch(batch_size=32).prefetch(buffer_size=tf.data.AUTOTUNE)
```



Prefetching



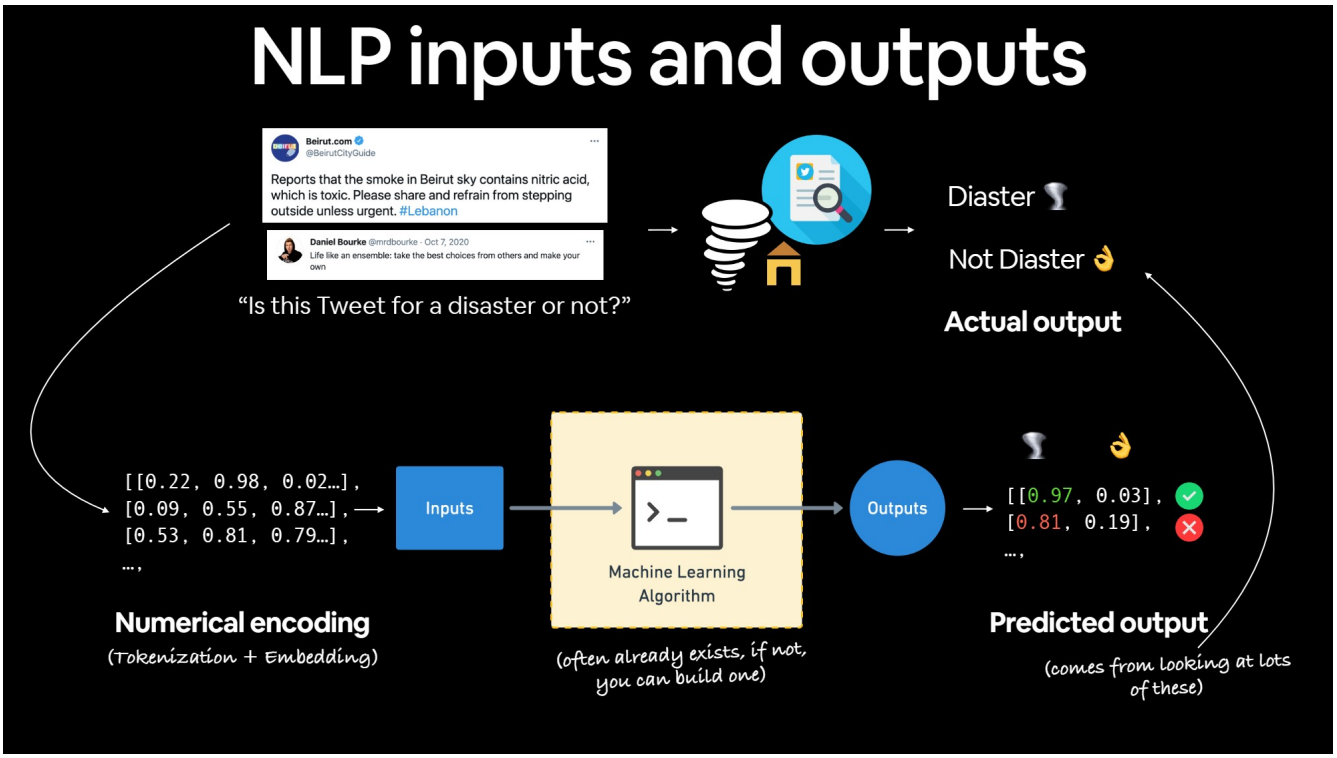
Source: Page 422 of Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow Book by Aurélien Géron

For loading data in the most performant way possible, see the TensorFlow documentation on Better performance with the tf.data API (https://www.tensorflow.org/guide/data_performance).

(Note: With larger models, *Model Checkpoint, earlyStopCallback*, and *ReduceLROnPlateau* callbacks are very useful)

5. NLP (Natural Language Processing)

- NLP problems are also referred to as *Sequence to Sequence Problems (seq2seq)* .
- Words are sometimes called ***Tokens*** in NLP.

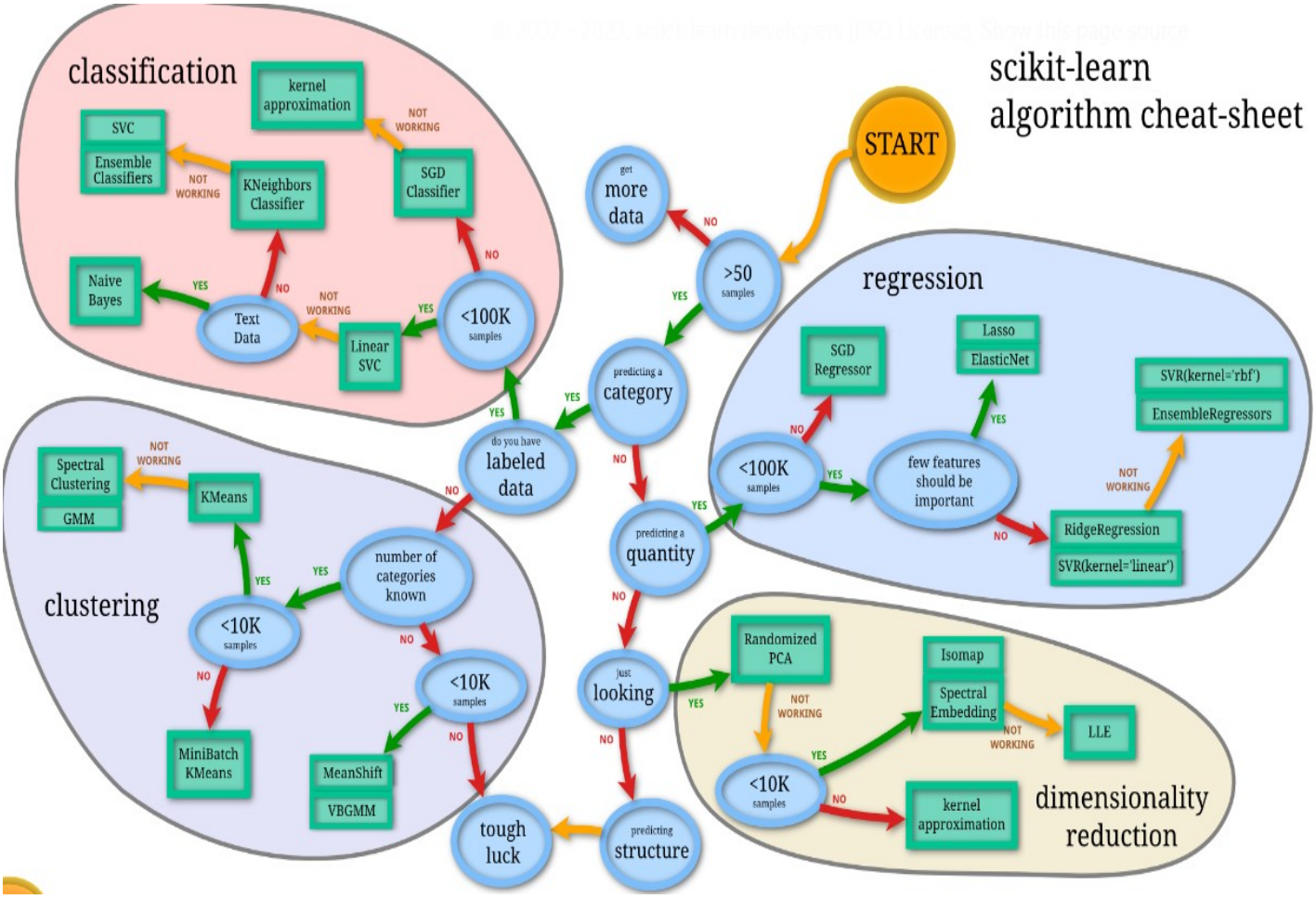


We'll go through several models in this chapter, each explaining key models of NLP problems:

Experiments we're running

Experiment Number	Model
0	Naive Bayes with TF-IDF encoder (baseline)
1	Feed-forward neural network (dense model)
2	LSTM (RNN)
3	GRU (RNN)
4	Bidirectional-LSTM (RNN)
5	1D Convolutional Neural Network
6	TensorFlow Hub Pretrained Feature Extractor
7	TensorFlow Hub Pretrained Feature Extractor (10% of data)

Notice that the first model is not a Deep Learning Model, this is because often times, before going full-on with deep learning we start with a machine-learning baseline, which can help us understand the problem and get us started so we can jump on more complex models, ***for this purpose we can use the scikit-learn machine learning map cheat-sheet:***


scikit-learn
algorithm cheat-sheet
START
get more data
NO
>50 samples
YES
predicting a category
YES
do you have labeled data
YES
<100K samples
NO
SGD Classifier
NOT WORKING
kernel approximation
NO
SVC
Ensemble Classifiers
NOT WORKING
KNeighbors Classifier
NO
Naive Bayes
YES
Text Data
NO
NOT WORKING
Linear SVC
classification
NO
predicting a quantity
YES
<100K samples
NO
SGD Regressor
YES
few features should be important
YES
Lasso
ElasticNet
NO
RidgeRegression
SVR(kernel='linear')
NOT WORKING
SVR(kernel='rbf')
EnsembleRegressors
regression
NO
just looking
YES
Randomized PCA
NOT WORKING
<10K samples
YES
Isomap
Spectral Embedding
NOT WORKING
LLE
NO
kernel approximation
dimensionality reduction
NO
predicting structure
tough luck
NO
number of categories known
YES
<10K samples
YES
KMeans
NOT WORKING
Spectral Clustering
GMM
NO
MiniBatch KMeans
<10K samples
YES
MeanShift
VBGMM
NO
clustering

5.1. NLP Datasets

A common problem in NLP datasets is an **unbalanced dataset:** *an unbalanced dataset* is a dataset where the number of data samples is not the same across different classes. This means one class has more data samples than the other class. For instance, one class could have 3000 samples, and the other may have only 300. The class with more data samples is known as the *majority class*, while the other one is known as the *minority class*.

When we train a model with an imbalanced dataset, the model tends to be biased towards the majority class. This can lead to inaccurate results and wrong predictions.

To handle imbalanced datasets, various techniques can be applied:

1.**Resampling**: This involves either oversampling the minority class or undersampling the majority class to balance the dataset. Oversampling involves replicating some of the observations from the minority class to increase its representation in the dataset.

2.**Class Weights**: In this method, the model is made to pay more attention to the minority class during training.

3.**Downsampling and Upweighting**: This technique involves reducing the number of samples in the majority class and increasing the importance of these samples during model training. These techniques help reduce model bias and enhance the performance of NLP models. However, it's always recommended to first try training on the true distribution of the dataset. If the model works well and generalizes, then no further action is needed. If not, then these techniques can be applied.

5.2. Converting Text Into Numbers in NLP Problems

In NLP there are basically two main concepts for turning text into numbers:

1. **Tokenization** - A straight mapping from word or character or sub-word to a numerical value. There are three main levels of tokenization:

    - **Word-level tokenization:** e.g. "I love TensorFlow" might result in "I" being `0`, "love" being `1` and "TensorFlow" being `2`. In this case, *every word in a sequence considered a single token.*

    - **Character-level tokenization:** such as converting the letters *A-Z* to values *1-26.* In this case, *every character in a sequence considered a single token.*

    - **Sub-word tokenization:** is in between word-level and character-level tokenization. It involves breaking invidual words into smaller parts and then converting those smaller parts into numbers. For example, "my favourite food is pineapple pizza" might become "my, fav, avour, rite, fo, oo, od, is, pin, ine, app, le, piz, za".

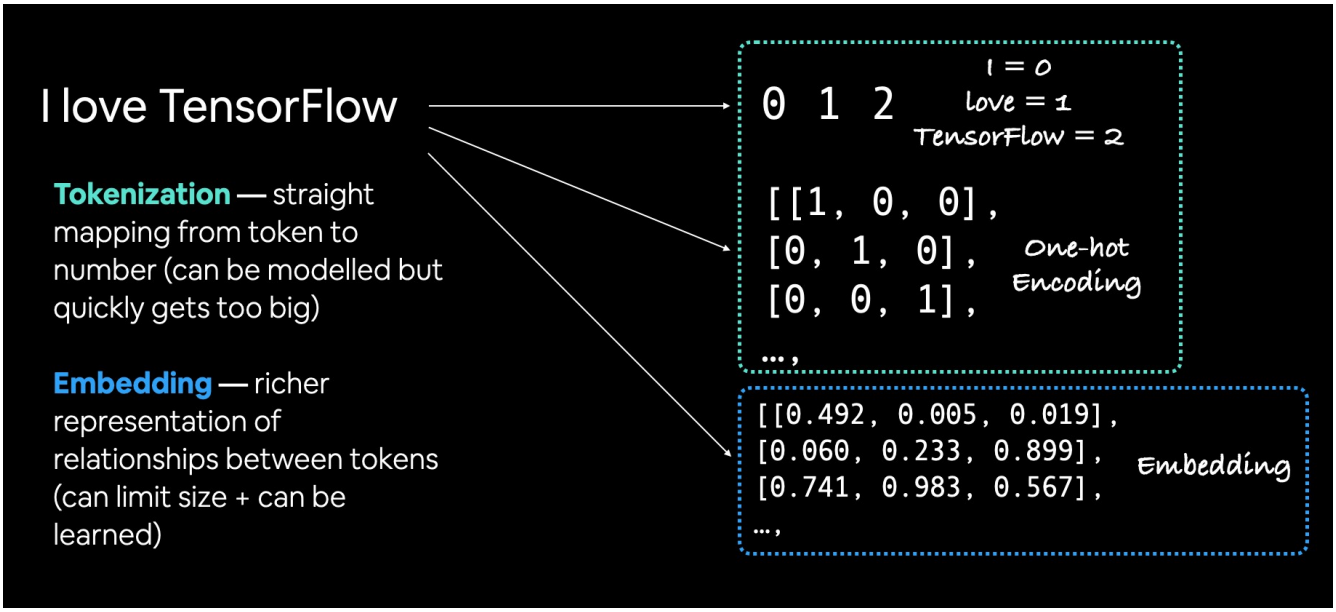
      After doing this, these sub-words would then be mapped to a numerical value. In this case, every word could be considered multiple **tokens**.

2. **Embeddings** - An embedding is a representation of natural language which can be learned. Representation comes in the form of a **feature vector**. For example, the word "dance" could be represented by the 5-dimensional vector `[-0.8547, 0.4559, -0.3332, 0.9877, 0.1112]`. It's important to note here, *the size of the feature vector is tuneable*. There are two ways to use embeddings:

    - **Create your own embedding** - Once your text has been turned into numbers (required for an embedding), you can put them through an embedding layer (such as `tf.keras.layers.Embedding`) and an embedding

representation will be learned during model training.

- **Reuse a pre-learned embedding** - Many pre-trained embeddings exist online. These pre-trained embeddings have often been learned on large corpuses of text (such as all of Wikipedia) and thus have a good underlying representation of natural language. You can use a pre-trained embedding to initialize your model and fine-tune it to your own specific task.



(Note: *embeddings* can become a layer in a NN, therefore it can be learned and get updated as the model gets trained)

5.2.1. Tokenization in TensorFlow

To tokenize our words in tensorflow we can use a helpful preprocessing layer: tf.keras.layers.TextVectorization | TensorFlow v2.14.0 **:**

The **TextVectorization** layer takes the following parameters:

- *max_tokens* - The maximum number of words in your vocabulary (e.g. 20000 or the number of unique words in your text), includes a value for OOV (out of vocabulary) tokens.

- *standardize* - Method for standardizing text. Default is "lower_and_strip_punctuation" which lowers text and removes all punctuation marks.

- *split* - How to split text, default is "whitespace" which splits on spaces.

- *ngrams* - How many words to contain per token split, for example, *ngrams=2* splits tokens into continuous sequences of 2.

- *output_mode* - How to output tokens, can be "int" (integer mapping), “binary” (one-hot encoding), "count" or "tf-idf". See documentation for more.

- *output_sequence_length* - Length of tokenized sequence to output. For example, if output_sequence_length=150, all tokenized sequences will be 150 tokens long. (Note: that no matter the size of the sequence that we pass to the layer it will always be of the specified size, if it’s smaller, it will auto-complete with zeros.)

- *pad_to_max_tokens* - Defaults to False, if True, the output feature axis will be padded to max_tokens even if the number of unique tokens in the vocabulary is less than max_tokens. Only valid in certain modes, see docs for more.

In the course, the instructor wants to set values for *max_tokens* and for the

output_sequence_length. For *max_tokens* some common values are: multiples of 10,000 or the exact number of unique words in your text. For *output_sequence_length* the instructor wants to use the average number of tokens per tweet.

Putting it all together:

In [16]:
```
# Find average number of tokens (words) in training Tweets
round(sum([len(i.split()) for i in train_sentences])/len(train_sentences))
```

Out[16]: 15

Now let's create another `TextVectorization` object using our custom parameters.

In [17]:
```
# Setup text vectorization with custom variables
max_vocab_length = 10000 # max number of words to have in our vocabulary
max_length = 15 # max length our sequences will be (e.g. how many words from a Tweet does our model see?)

text_vectorizer = TextVectorization(max_tokens=max_vocab_length,
                                    output_mode="int",
                                    output_sequence_length=max_length)
```

Now, if we read the documentation for this layer we reach a part that says:

"The vocabulary for the layer must be either supplied on construction or learned via `adapt()`. When this layer is adapted, it will analyze the dataset, determine the frequency of individual string values, and create a vocabulary from them. This vocabulary can have unlimited size or be capped, depending on the configuration options for this layer; if there are more unique values in the input than the maximum vocabulary size, the most frequent terms will be used to create the vocabulary. "

Which is what we do here:

In [18]:
```
# Fit the text vectorizer to the training text
text_vectorizer.adapt(train_sentences)
```

Training data mapped! Let's try our `text_vectorizer` on a custom sentence (one similar to what you might see in the training data).

In [19]:
```
# Create sample sentence and tokenize it
sample_sentence = "There's a flood in my street!"
text_vectorizer([sample_sentence])
```

Out[19]:
```
<tf.Tensor: shape=(1, 15), dtype=int64, numpy=
array([[264,   3, 232,   4,  13, 698,   0,   0,   0,   0,   0,   0,   0,
          0,   0]])>
```

5.2.2. Embedding in TensorFlow

Now let's see at how to create an embedding layer. For this we'll use: tf.keras.layers.Embedding | TensorFlow v2.14.0 which takes as parameters:

- *input_dim* - The size of the vocabulary(e.g. len(text_vectorizer.get_vocabulary()).
- *output_dim* - The size of the output embedding vector, for example, a value of 100 outputs a feature vector of size 100 for each word.
- *embeddings_initializer* - How to initialize the embeddings matrix, default is "uniform" which randomly initalizes embedding matrix with uniform distribution. This can be changed for using pre-learned embeddings.
- *input_length* - Length of sequences being passed to embedding layer.

```
tf.random.set_seed(42)
from tensorflow.keras import layers

embedding = layers.Embedding(input_dim=max_vocab_length, # set input shape
                             output_dim=128, # set size of embedding vector
                             embeddings_initializer="uniform", # default, intialize randomly
                             input_length=max_length, # how long is each input
                             name="embedding_1")

embedding
```

But remember, we need to pass it the text that we want to embed *in numerical format:*

```
# Embed the random sentence (turn it into numerical representation)
sample_embed = embedding(text_vectorizer([random_sentence]))
```

5.3. Model 0 (baseline) – Naive Bayes with TF-IDF encoder

In [25]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

# Create tokenization and modelling pipeline
model_0 = Pipeline([
                    ("tfidf", TfidfVectorizer()), # convert words to numbers using tfidf
                    ("clf", MultinomialNB()) # model the text
])

# Fit the pipeline to the training data
model_0.fit(train_sentences, train_labels)
```

In [26]:
```
baseline_score = model_0.score(val_sentences, val_labels)
print(f"Our baseline model achieves an accuracy of: {baseline_score*100:.2f}%")
```

```
Our baseline model achieves an accuracy of: 79.27%
```

```
def calculate_results(y_true, y_pred):
  """
  Calculates model accuracy, precision, recall and f1 score of a binary classification model.

  Args:
  -----
  y_true = true labels in the form of a 1D array
  y_pred = predicted labels in the form of a 1D array

  Returns a dictionary of accuracy, precision, recall, f1-score.
  """
  # Calculate model accuracy
  model_accuracy = accuracy_score(y_true, y_pred) * 100
  # Calculate model precision, recall and f1 score using "weighted" average
  model_precision, model_recall, model_f1, _ = precision_recall_fscore_support(y_true, y_pred, average="weighted")
  model_results = {"accuracy": model_accuracy,
                   "precision": model_precision,
                   "recall": model_recall,
                   "f1": model_f1}
  return model_results
```

In [29]:

```
# Get baseline results
baseline_results = calculate_results(y_true=val_labels,
                                     y_pred=baseline_preds)
baseline_results
```

Out[29]:

```
{'accuracy': 79.26509186351706,
 'precision': 0.8111390004213173,
 'recall': 0.7926509186351706,
 'f1': 0.7862189758049549}
```

5.4. Model 1: Feed-Forward Network (Dense)

In [31]:

```
# Build model with the Functional API
from tensorflow.keras import layers
inputs = layers.Input(shape=(1,), dtype="string") # inputs are 1-dimensional strings
x = text_vectorizer(inputs) # turn the input text into numbers
x = embedding(x) # create an embedding of the numerized numbers
x = layers.GlobalAveragePooling1D()(x) # lower the dimensionality of the embedding (try running the model without this laye
outputs = layers.Dense(1, activation="sigmoid")(x) # create the output layer, want binary outputs so use sigmoid activation
model_1 = tf.keras.Model(inputs, outputs, name="model_1_dense") # construct the model
```

There's a key point here, if we don't do an **average pooling** to reduce the dimensionality of our tensor, our model will output a prediction probability for every token in each sentence:

```
[75] model_1_pred_probs[0]

     array([[0.42709357],
            [0.42709357],
            [0.42709357],
            [0.20030434],
            [0.5064465 ],
            [0.42709357],
            [0.42709357],
            [0.42709357],
            [0.10063491],
            [0.4038509 ],
            [0.42709357],
            [0.94114774],
            [0.03610222],
            [0.42709357],
            [0.3464149 ]], dtype=float32)
```

Our model will perform poorly this way (around 64% in this case), with *average pooling* our model performs at 79 %.

In [33]:
```
# Get a summary of the model
model_1.summary()
```

```
Model: "model_1_dense"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 1)]               0

 text_vectorization_1 (TextV  (None, 15)               0
 ectorization)

 embedding_1 (Embedding)     (None, 15, 128)           1280000

 global_average_pooling1d (G  (None, 128)              0
 lobalAveragePooling1D)

 dense (Dense)               (None, 1)                 129

=================================================================
Total params: 1,280,129
Trainable params: 1,280,129
Non-trainable params: 0
_________________________________________________________________
```

Most of the trainable parameters are contained within the embedding layer. Recall we created an embedding of size 128 (output_dim=128) for a vocabulary of size 10,000 (input_dim=10000), hence the 1,280,000 trainable parameters.

5.4.1. Visualizing Embeddings

A helpful tool for visualizing and understanding word embeddings is: http://projector.tensorflow.org/. To do so we'll need the weights and the metadata:

```
weights = model.get_layer('embedding').get_weights()[0]
vocab = vectorize_layer.get_vocabulary()
```

Write the weights to disk. To use the Embedding Projector, you will upload two files in tab separated format: a file of vectors (containing the embedding), and a file of meta data (containing the words).

```
out_v = io.open('vectors.tsv', 'w', encoding='utf-8')
out_m = io.open('metadata.tsv', 'w', encoding='utf-8')

for index, word in enumerate(vocab):
  if index == 0:
    continue  # skip 0, it's padding.
  vec = weights[index]
  out_v.write('\t'.join([str(x) for x in vec]) + "\n")
  out_m.write(word + "\n")
out_v.close()
out_m.close()
```

If you are running this tutorial in Colaboratory, you can use the following snippet to download these files to your local machine (or use the file browser, *View -> Table of contents -> File browser*).

```
try:
  from google.colab import files
  files.download('vectors.tsv')
  files.download('metadata.tsv')
except Exception:
  pass
```

(Code from: Word embeddings | Text | TensorFlow)

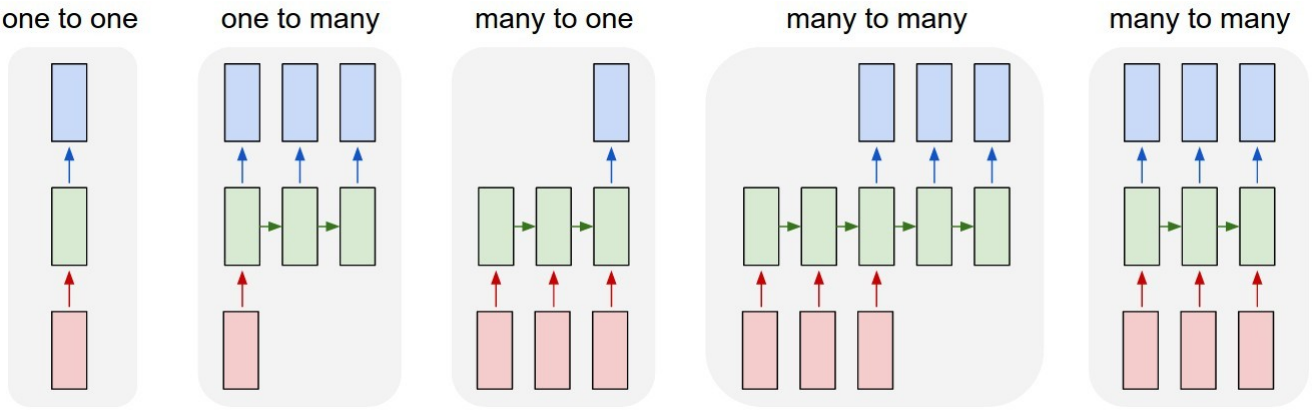
Once you've downloaded the embedding vectors and metadata, you can visualize them using Embedding Vector tool:

1. Go to http://projector.tensorflow.org/
2. Click on "Load data"
3. Upload the two files you downloaded (embedding_vectors.tsv and embedding_metadata.tsv)
4. Explore

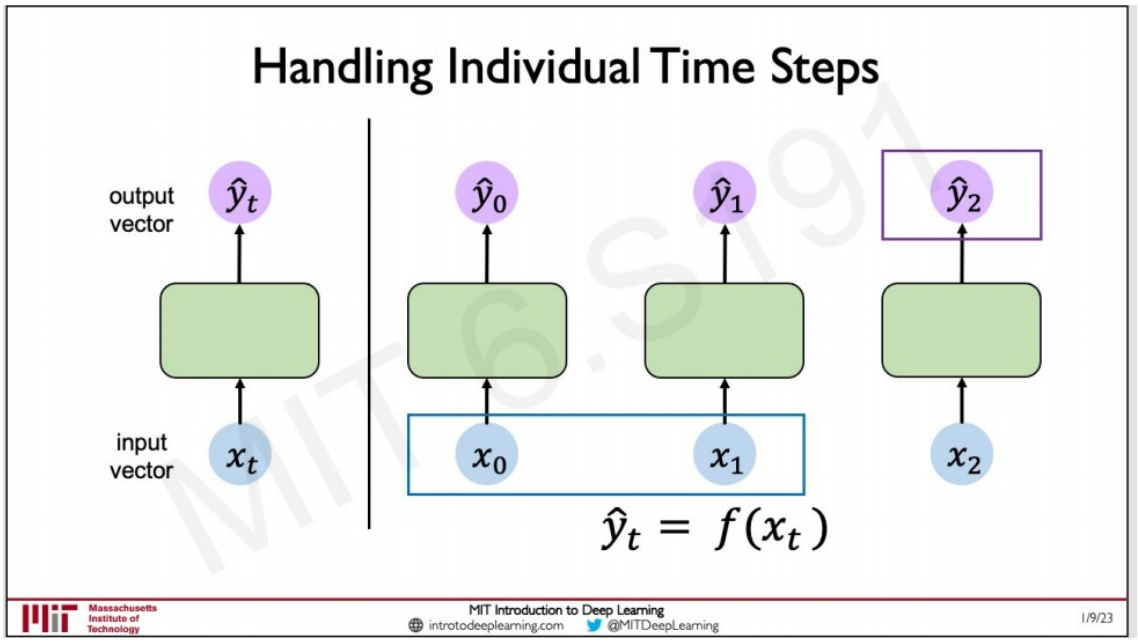
5.5. RNN's (Recurrent neural networks)

(Note: All the content in this particular chapter is from: (MIT Intro to DL, Lecture 2, Understanding LSTM Networks -- colah's blog, The Unreasonable Effectiveness of Recurrent Neural Networks (karpathy.github.io))

The key motivation to use RNNs: are to handle **sequential information** (the points in the dataset are dependent on the other points in the dataset) such as text, audio, etc. Feed-forward networks (*Also known as multilayer perceptrons (MLPs))* are suited for problems for **independent data** (each data point is independent of other data points).

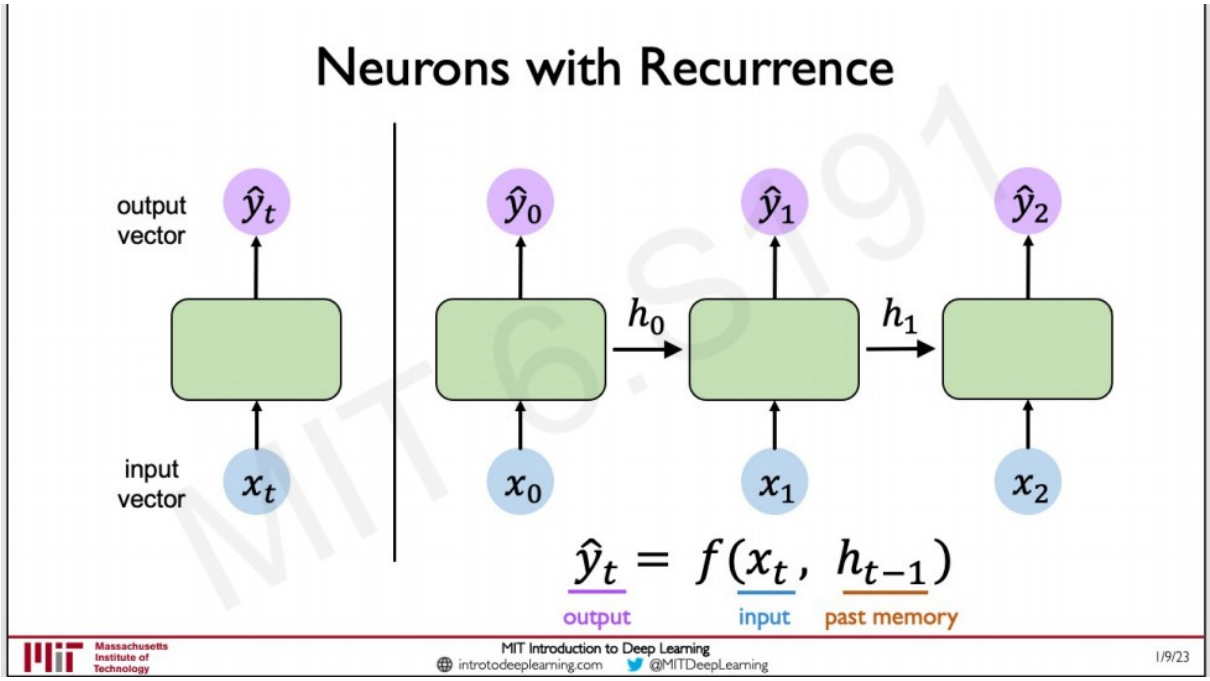


For example, say that we have a sequence, if we were to try to apply a perceptron model to sequential information it would look something like this:



The obvious answer as to why this wouldn't work is that *the output at timestep 2 for example ($\boldsymbol{X}_t$ represents each time step) could depend on the inputs that were at previous time steps if this was a temporal sequence with temporal dependencies.*

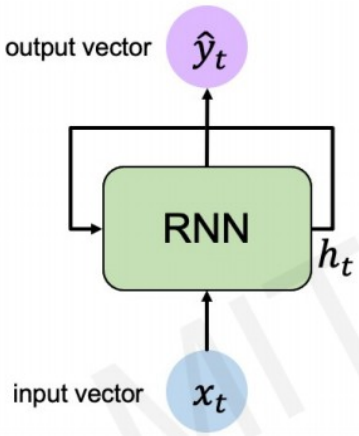
We can begin to think of a solution though, how about we pass the computation done by the network at one particular timesteps to the following timesteps, this "computation" is referred to as the **internal state**:



As you can see now, the output on a particular time step no longer depends ONLY on the input in a particular time step, but ALSO on the past memory, the internal state of the previous layer.

This is often represented as a loop:

Recurrent Neural Networks (RNNs)



Apply a **recurrence relation** at every time step to process a sequence:

$$h_t = f_W(x_t, h_{t-1})$$

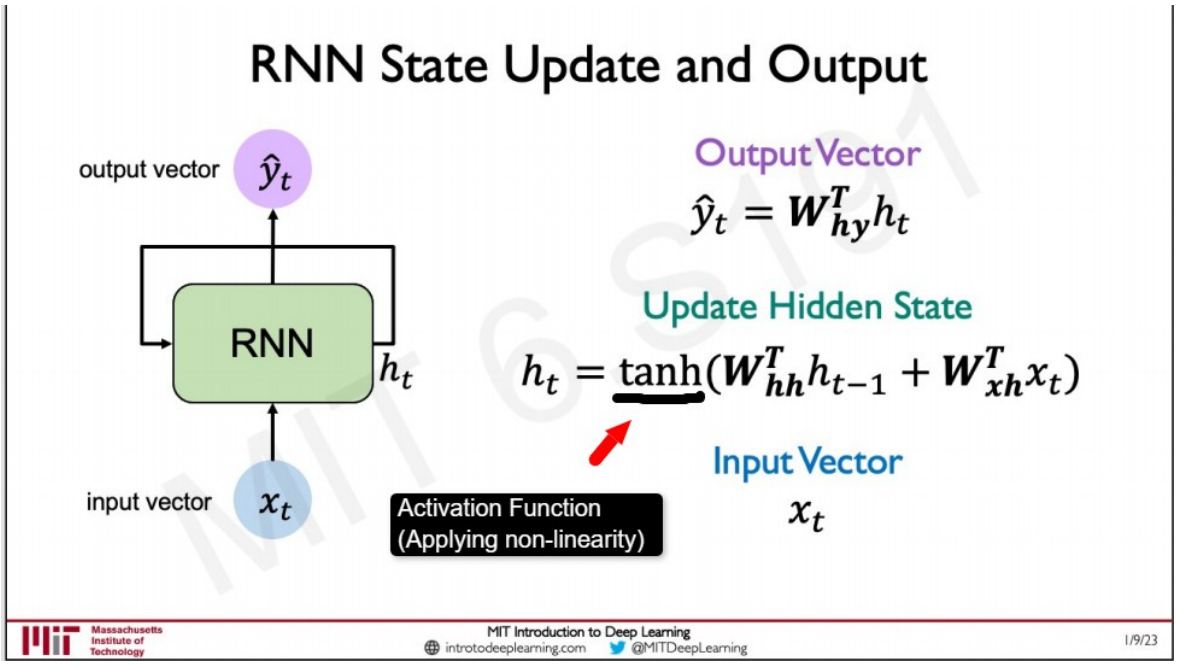
cell state | function with weights W | input | old state

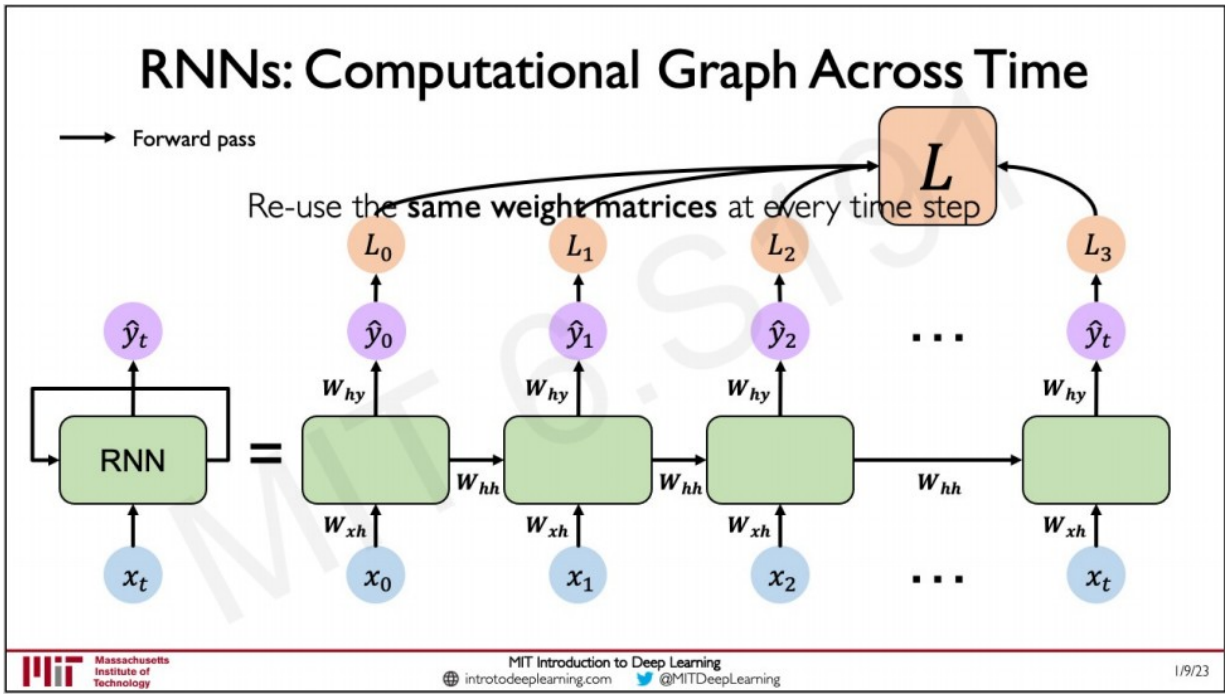
Note: the same function and set of parameters are used at every time step

RNNs have a **state**, h_t, that is updated **at each time step** as a sequence is processed

5.5.1. Unfolding RNNs

How are we generating the output prediction itself?





5.5.2. Sequence Modelling: Design Criteria

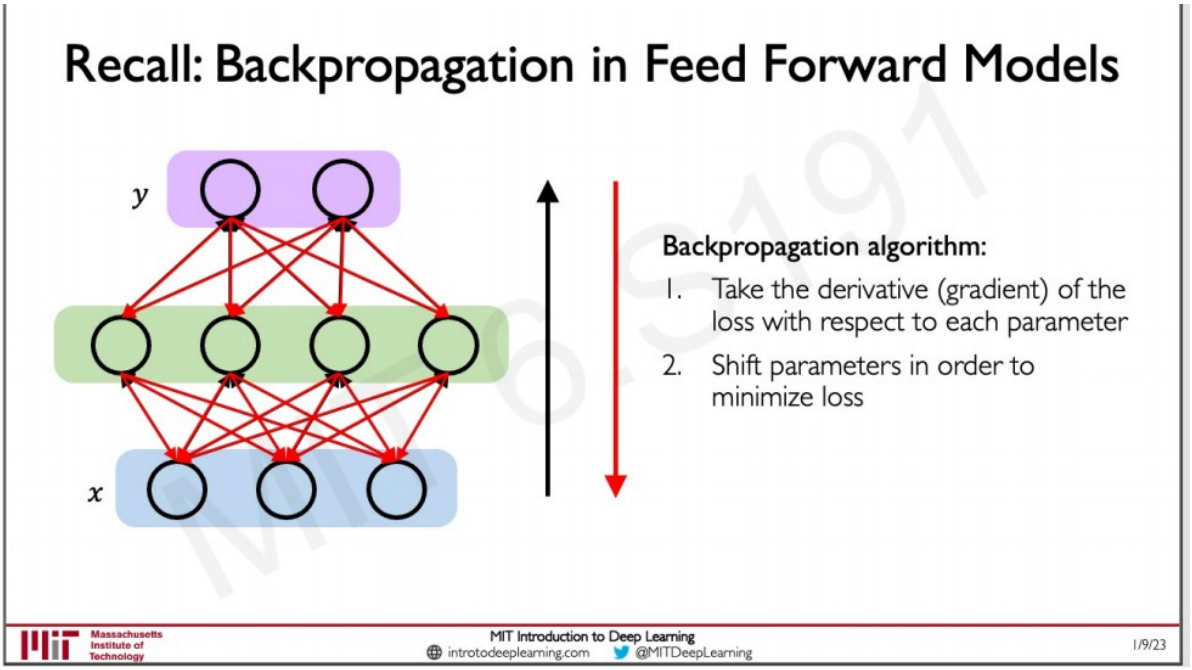
What are the design requirements that the RNN needs to fulfill to handle sequential data effectively:

- *Handle **variable-length** sequences* - Sequences may be short or long, our model needs to handle that.
- *Track **long-term** dependencies*
- *Maintain information about **order*** - Sequences depend on prior inputs and the specific order of things we see affects our result.
- ***Share parameters** across the sequence (**Parameter Sharing**)* - Given one set of weights, that set of weights should be able to apply to different time steps in the sequence and still result in a meaningful prediction.

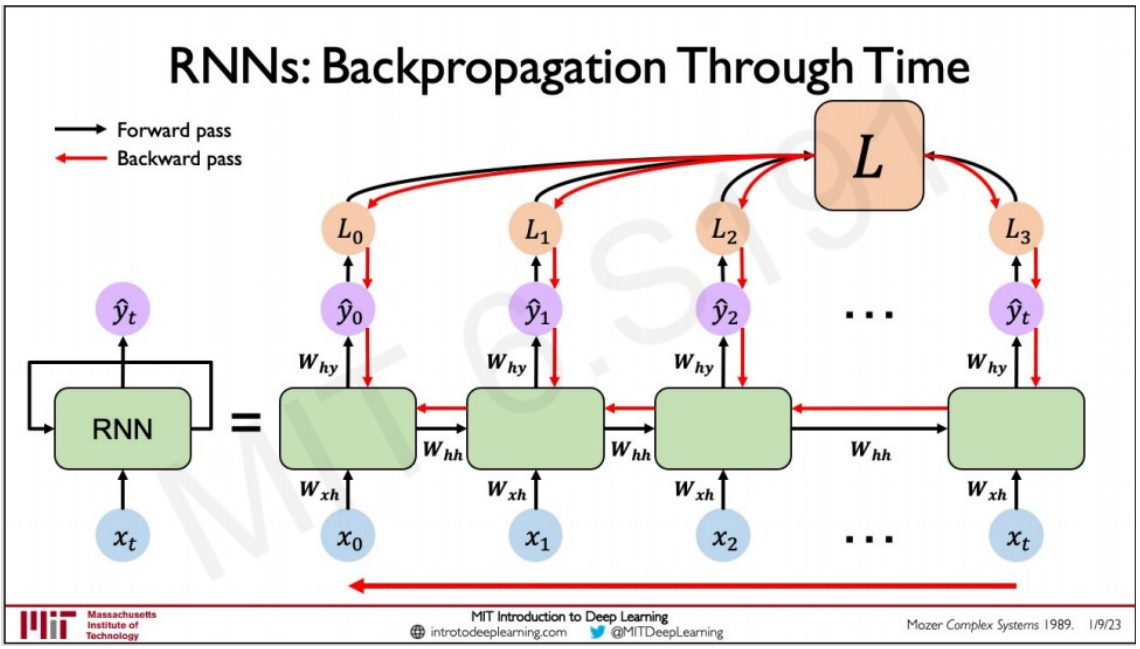
5.5.3. Backpropagation Through Time (BPTT)

How do we actually train and learn the weights in the RNN? This is done with a technique called Backpropagation through time.

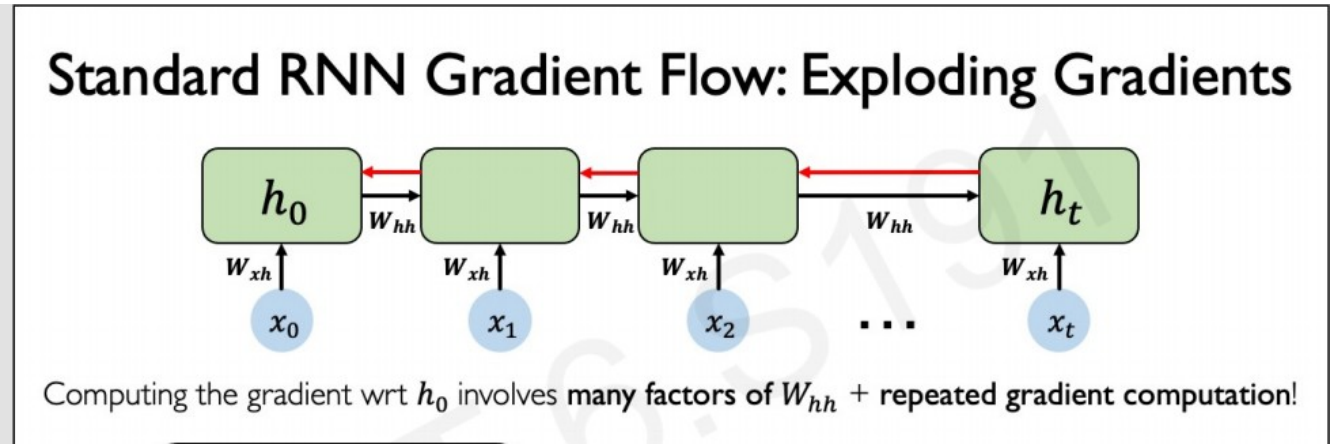
Recall that vanilla backpropagation takes the prediction and backpropagates gradients to adjust the parameters:



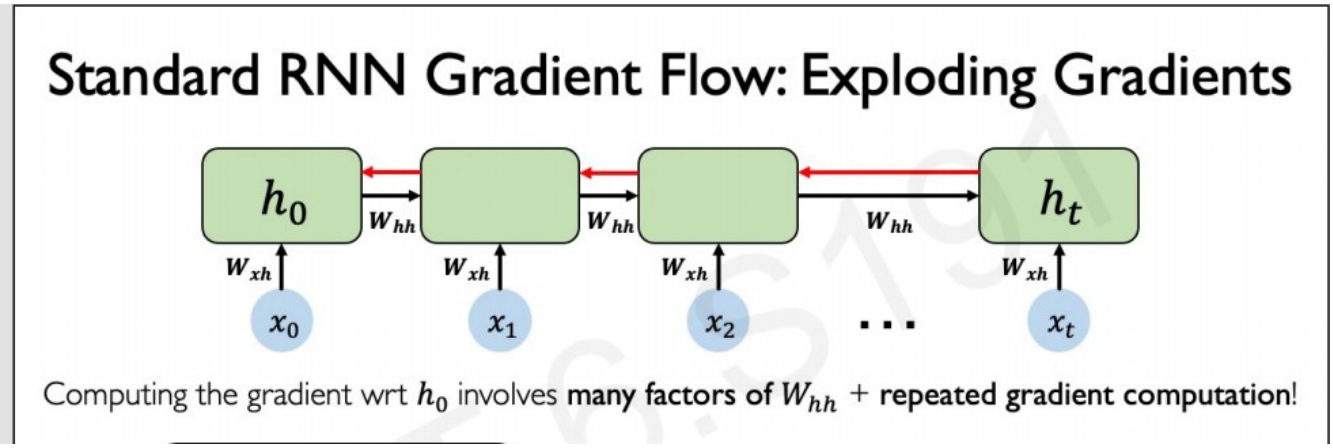
Now, in the case of RNNs, we have to backpropagate the loss through each of the individual time steps and after that we do that we must backpropagate it across all time steps (from time t to the beginning of the sequence) as illustrated below:

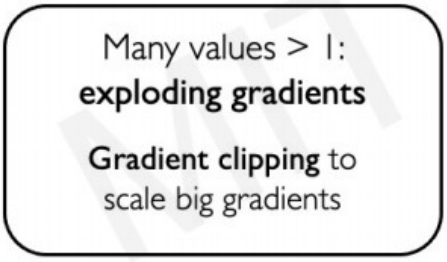


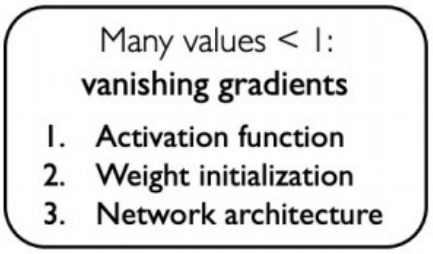
The tricky part about this is that looking at how gradient flows across the RNN, we notice that the algorithm takes A LOT of matrix multiplications, this bring us to potential gradient problems.



5.5.4. Gradient Problems

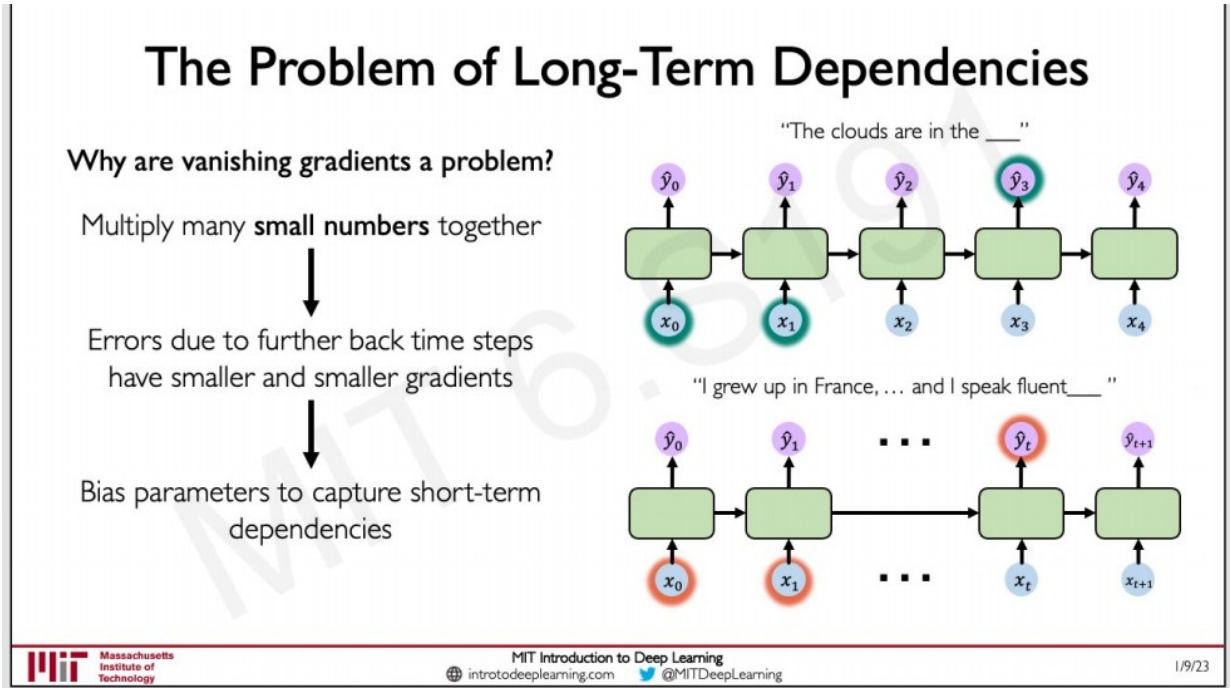






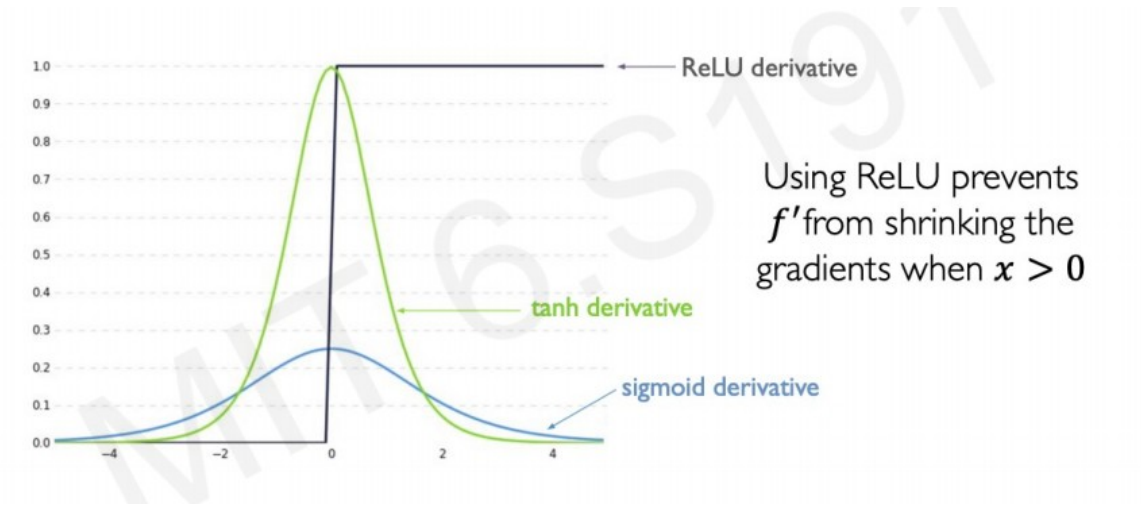
If this weight matrix *W* is really really big, our gradients can explode (meaning they get really big) making it infeasible to train our network effectively. To solve this we do **gradient clipping,** which effectively scales back those big gradients.

Conversely, our weight matrices can get very very small, resulting in a **vanishing gradient**, in cases where the information that we need is not far behind in the sequence there's no problem, but alternatively, if we need a long-term dependency to determine the output at a particular time step we run into problems, as illustrated below:



As illustrated on the previous page, we have three tricks that we can use to try to mitigate this:

1. **Change Activation Function:**



2. **Weight Initialization:**

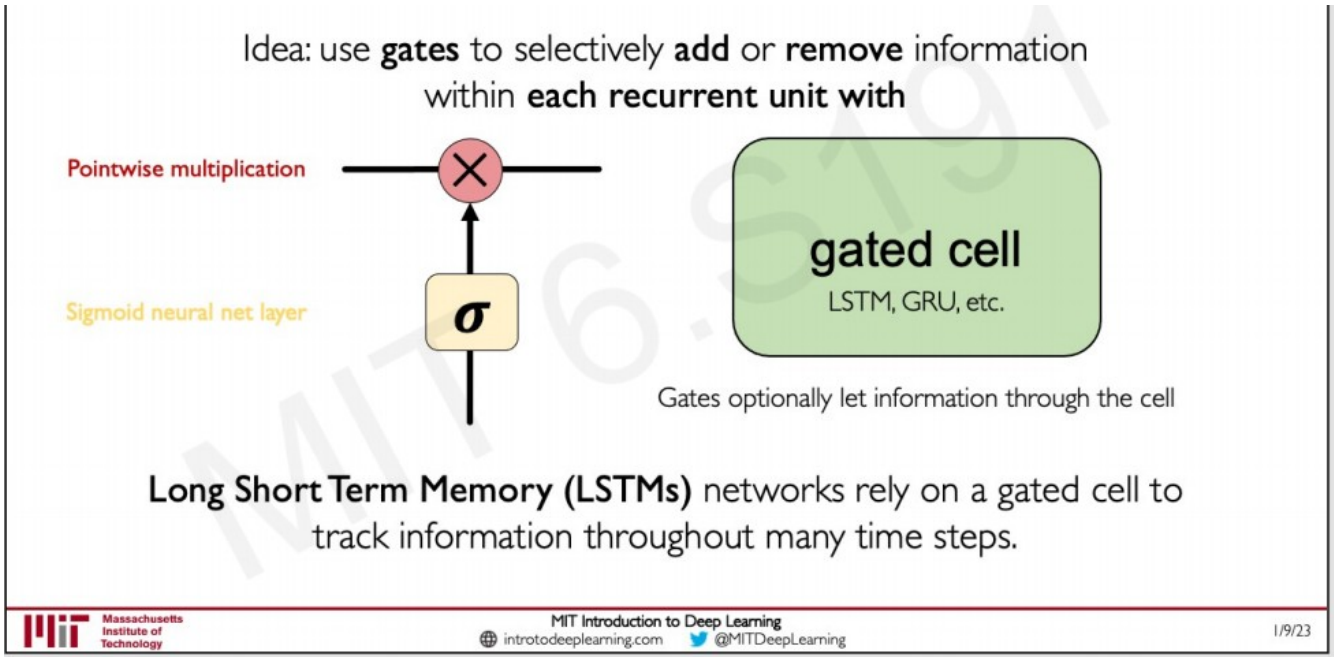
Initialize **weights** to identity matrix

Initialize **biases** to zero

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

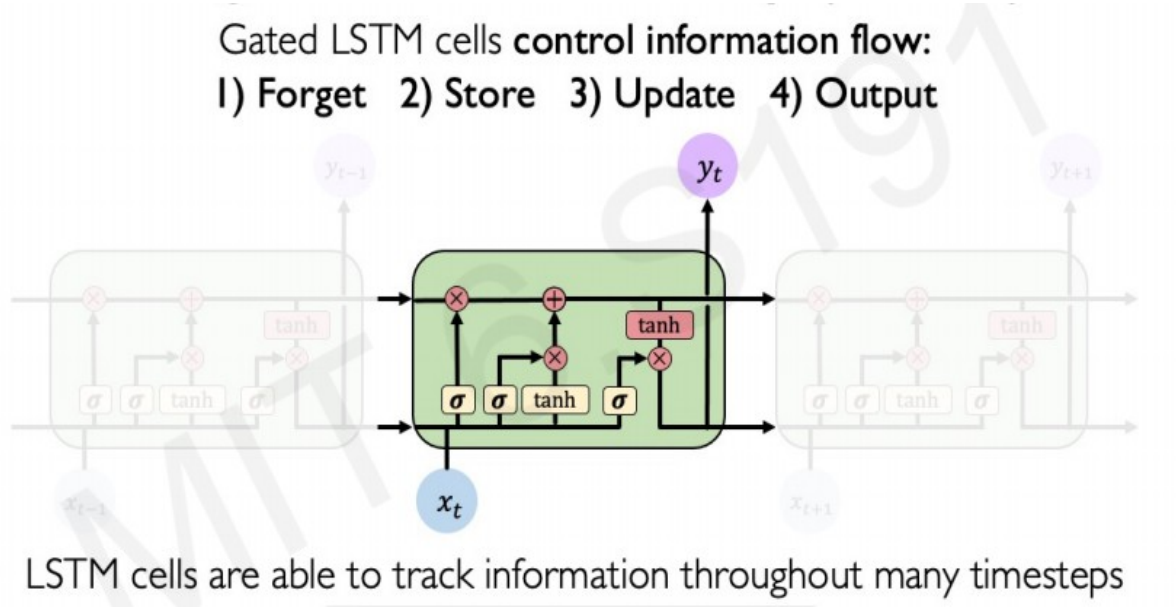
This helps prevent the weights from shrinking to zero.

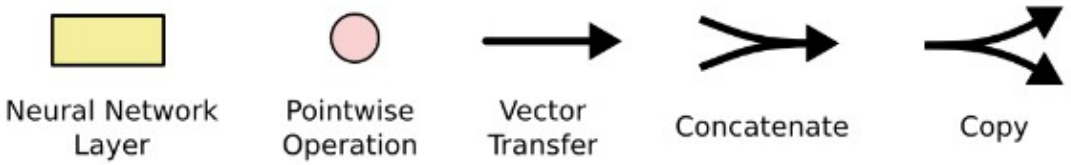
3. **Gated Cells:** The most robust solution to the *vanishing gradient* problem. LSTM are the most commonly used.



5.5.5. LSTMs (Long Short Term Memory Networks)

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997).





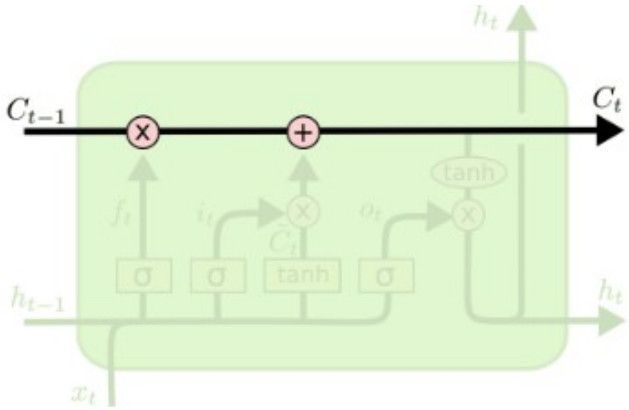
In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.

LSTMs Key Concepts:

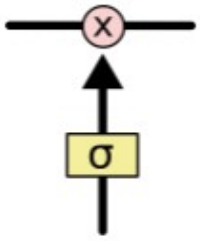
1. Maintain a **cell state**
2. Use **gates** to control the **flow of information**
    - **Forget** gate gets rid of irrelevant information
    - **Store** relevant information from current input
    - Selectively **update** cell state
    - **Output** gate returns a filtered version of the cell state
3. Backpropagation through time with partially **uninterrupted gradient flow**

Step-by-Step LSTM Walk through:

1. The key to LSTMs is the **cell state**, <u>the horizontal line running through the top of the diagram:</u>



2. The LSTM has the ability to remove or add information to the cell state, carefully regulated by structures called **gates**. *Gates* are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

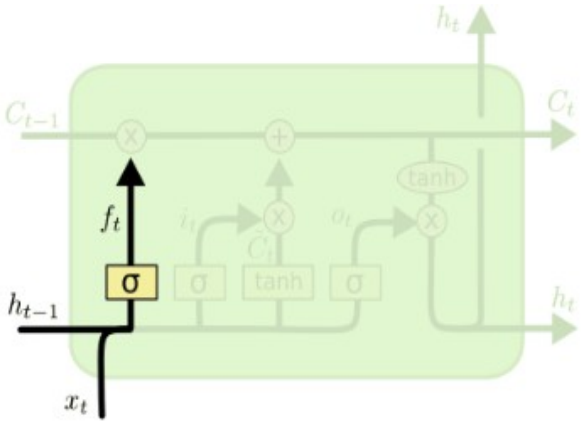


The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. An LSTM has three of these gates, to protect and control the cell state.

- **Forget Gate Layer:** The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a *sigmoid layer* called the **"forget gate layer".** It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 represents "completely keep this" while a 0 represents "completely get rid of this."

  For example, in a language model trying to predict the next word based on all the previous ones, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.

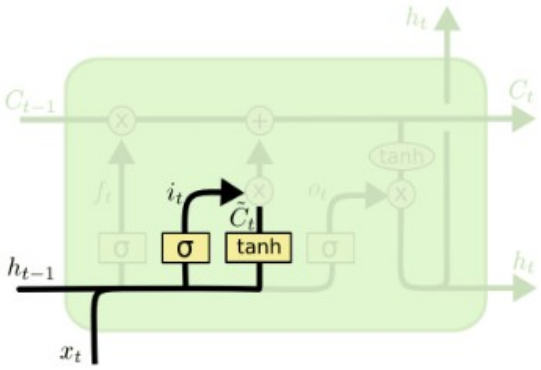
(SEE IMAGE ON NEXT PAGE)



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- **Store layers:** The next step is to decide what new information we're going to store in the cell state. *This has two parts*. First, a *sigmoid* layer called the **"input gate layer"** decides which values we'll update. Next, a *tanh* layer creates a vector of new candidate values: $\check{C}_t$ that could be added to the state.

  In the next step, we'll combine these two to create an **update to the state.** In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.

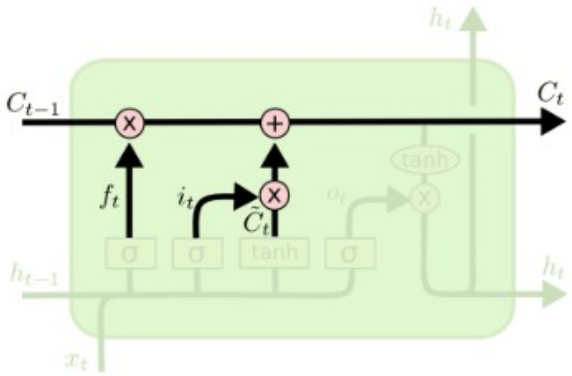


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

- **Update State:** It's now time to update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it.

  We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \check{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.

  In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.

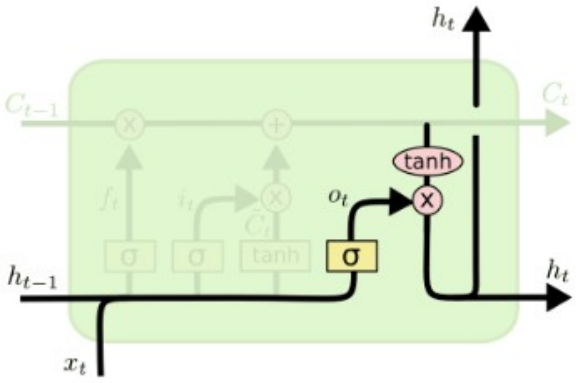


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- **Output gate:** Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version.

  First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through *tanh* (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

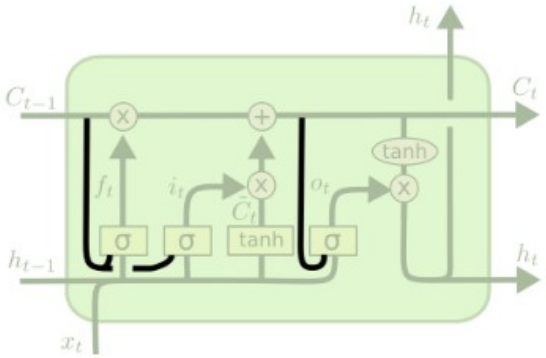
For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

5.5.6. Variants of LSTMs

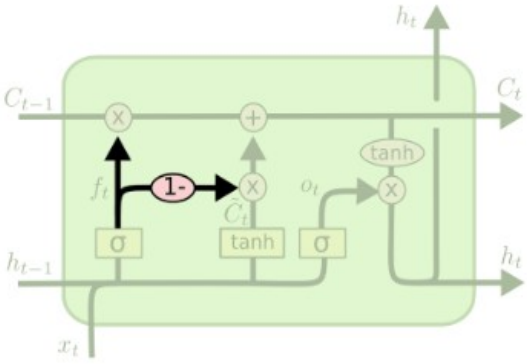
- **Peephole connections:** One popular LSTM variant, introduced by Gers & Schmidhuber (2000), is adding **"peephole connections".** This means that we let the gate layers look at the cell state.



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] + b_o\right)$$

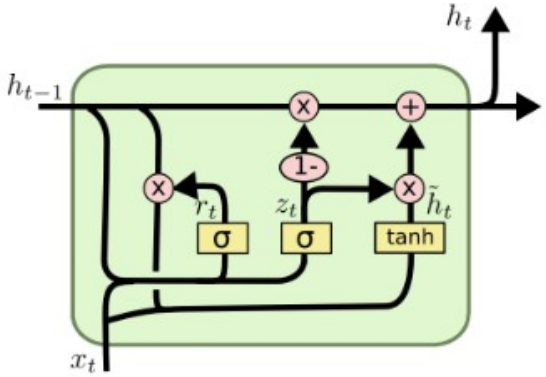
The above diagram adds peepholes to all the gates, but many papers will give some peepholes and not others.

- **Coupled forget and input gates:** Another variation is to use **coupled forget and input gate**s. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when we're going to input something in its place. We only input new values to the state when we forget something older.



$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

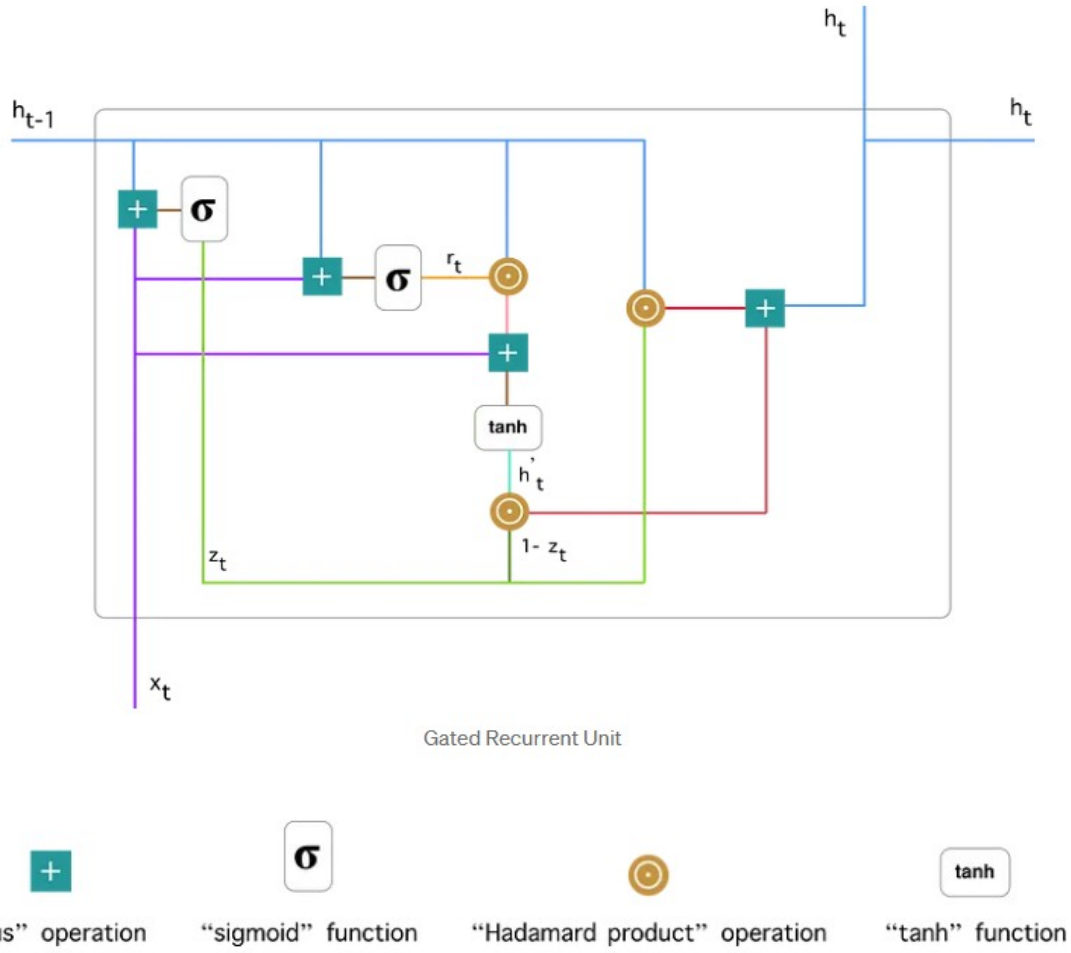
- **Gated Recurrent Unit (GRU):** A slightly more dramatic variation on the LSTM is the **Gated Recurrent Unit, or GRU**, introduced by Cho, et al. (2014). It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

GRUs are generally simpler and computationally more efficient than LSTMs, which makes them faster to train and more suitable for smaller datasets. However, LSTMs are generally more expressive and can capture more complex patterns in the data. This makes them more suitable for larger datasets and tasks that require more context understanding

Let's go into more detail about **GRUs**:



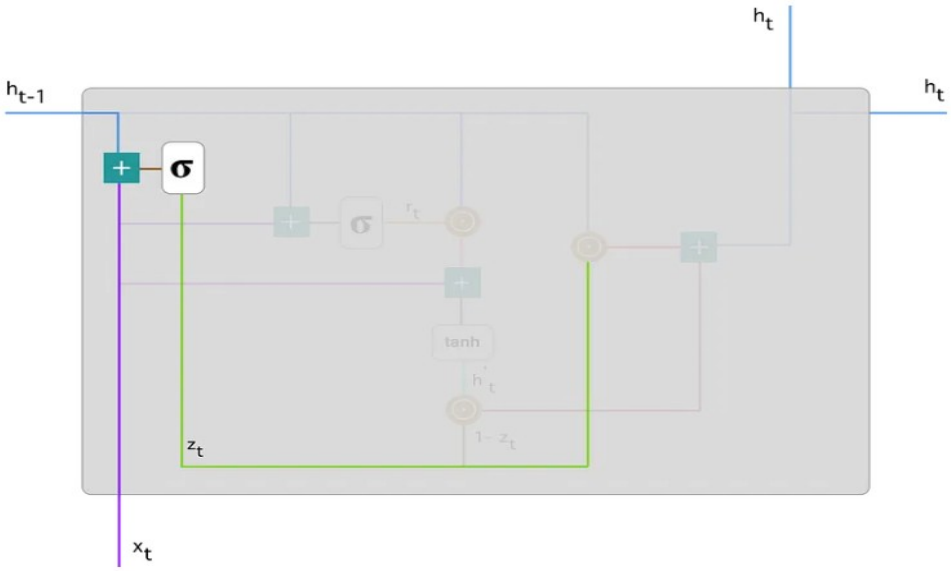
(Note: The Hadamard product is just like a regular dot product, but it operates on identically shaped matrices and produces a third matrix of the same dimensions. Instead of summing the multiplication of the corresponding components we just leave them as they are, resulting in a matrix of the same size as the others).

- **Update Gate:** We start with calculating the **update gate z_t for time step t** using the formula:

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

When x_t is plugged into the network unit, it is multiplied by its own weight ***W(z)***. The same goes for h_{t-1} which holds the information for the previous ***t-1*** units and is multiplied by its own weight ***U(z)***.

Both results are added together and a sigmoid activation function is applied to squash the result between 0 and 1. Following the above schema, we have:

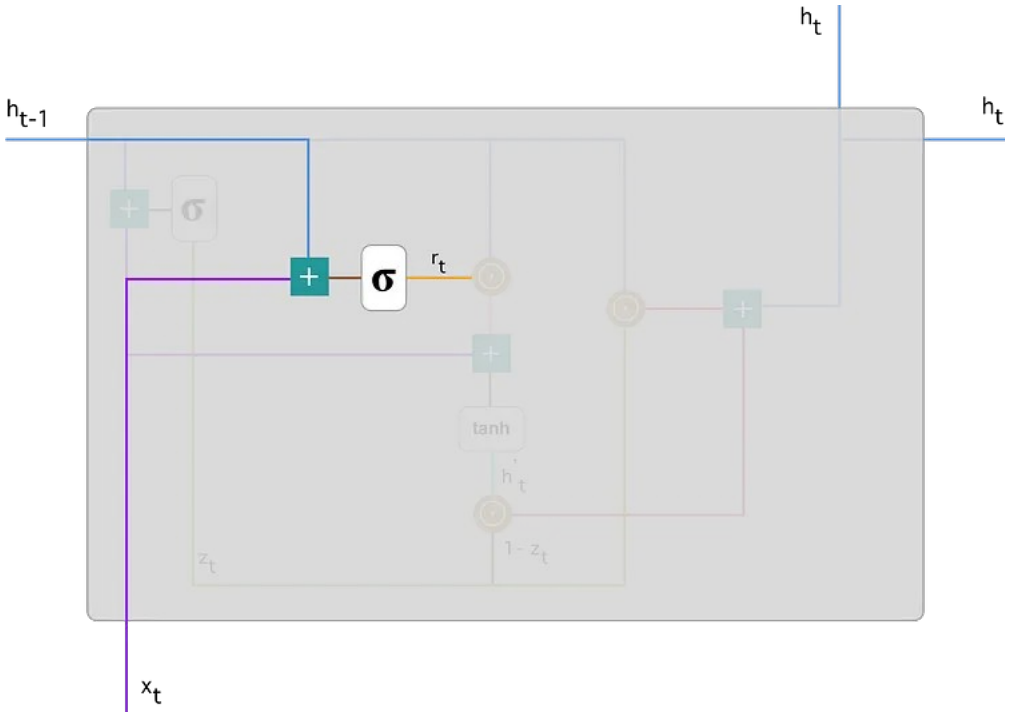


The update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future.

- **Reset Gate:** Essentially, **this gate is used from the model to decide how much of the past information to forget.** To calculate it, we use:

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

This formula is the same as the one for the update gate. The difference comes in the weights and the gate's usage.



- **Current Memory Content:** Let's see how exactly the gates will affect the final output. First, we start with the usage of the reset gate. **We introduce a new memory content which will use the reset gate to store the relevant information from the past.** It is calculated as follows:

$$h_t' = \tanh(Wx_t + r_t \odot Uh_{t-1})$$

1. Multiply the input x_t with a weight matrix W and h_{t-1} with a weight matrix U.

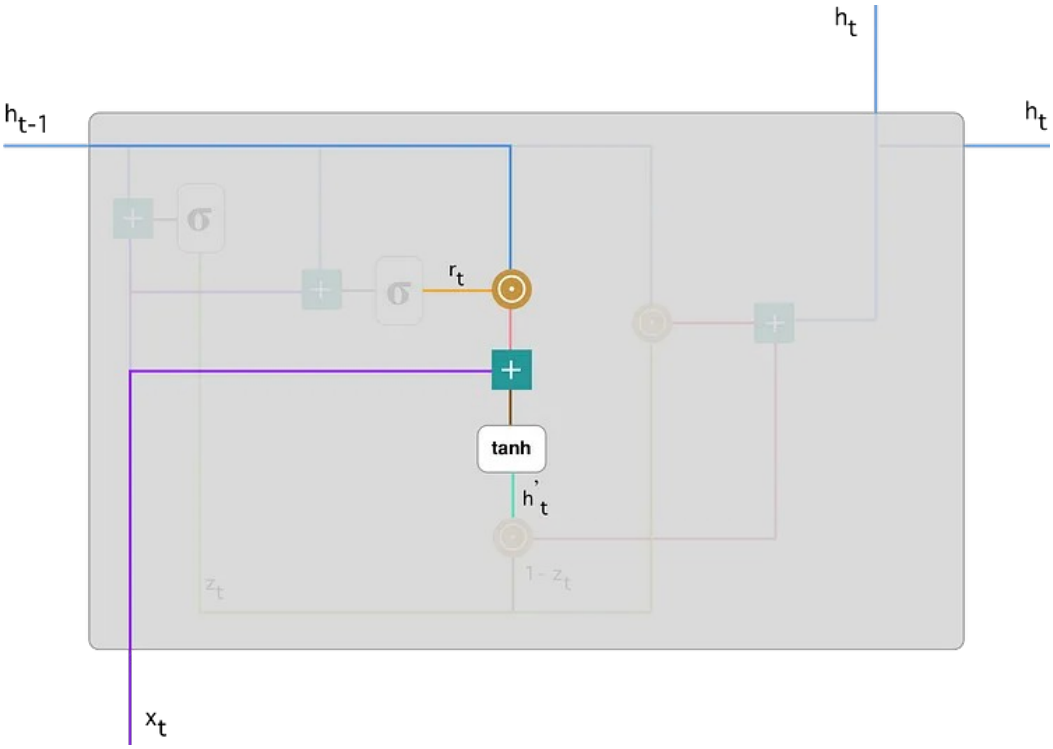
2. Calculate the Hadamard (element-wise) product between the reset gate r_t and Uh_{t-1}. That will determine what to remove from the previous time steps.

   Example: Let's say we have a sentiment analysis problem for determining one's opinion about a book from a review he wrote. The text starts with "This is a fantasy book which illustrates…" and after a couple paragraphs ends with "I didn't quite enjoy the book because I think it captures too many details." To determine the overall level of satisfaction from the book we only need the last part of the review. *In that case as the neural network approaches to the end of the text it will learn to assign* r_t *vector close to 0, washing out the past and focusing only on the last sentences.*

3. Sum up the results of step 1 and 2.

4. Apply the nonlinear activation function *tanh*.

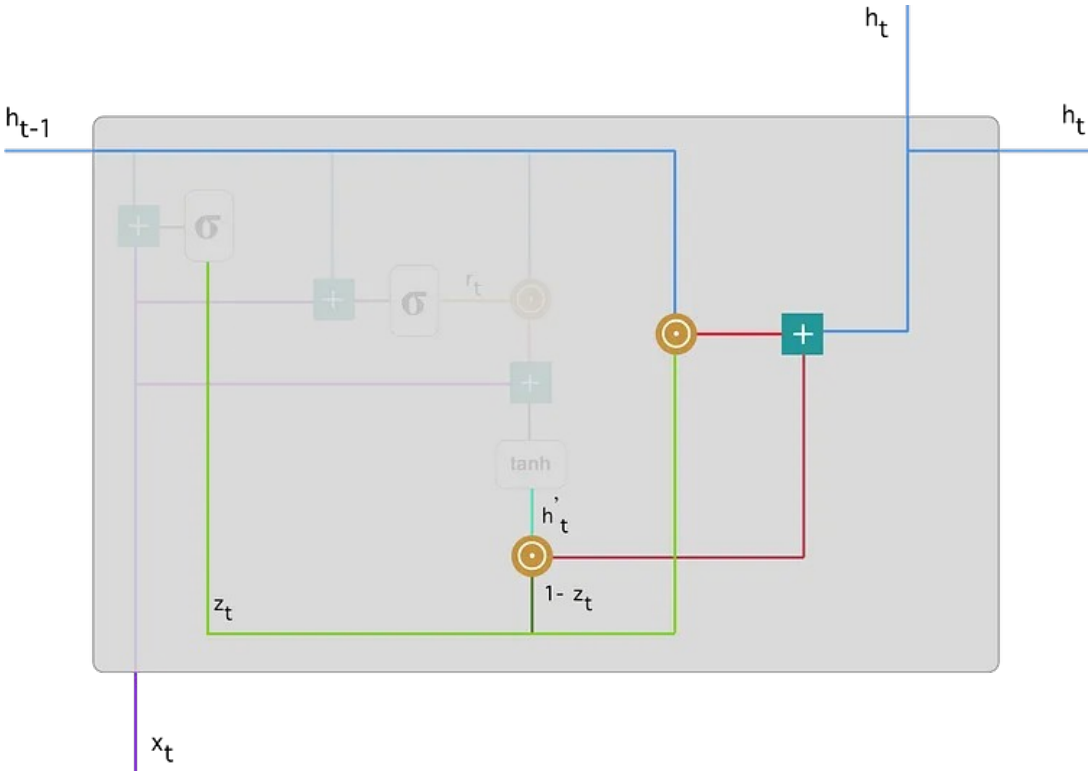
You can clearly see the steps here:



- **Final memory at current time step:** As the last step, the network needs to calculate h_t — vector which holds information for the current unit and passes it down to the network. In order to do that the update gate is needed. It determines what to collect from the current memory content — h'_t and what from the previous steps — h_{t-1}. That is done as follows:

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h'_t$$

1. Apply element-wise multiplication to the update gate z_t and h_{t-1}.
2. Apply element-wise multiplication to $(1- z_t)$ and h'_t.
3. Sum the results from step 1 and 2.



Let's bring up the example about the book review. This time, the most relevant information is positioned in the beginning of the text. The model can learn to set the vector z_t close to 1 and keep a majority of the previous information. Since z_t will be close to 1 at this time step, $1\text{-}z_t$ will be close to 0 which will ignore big portion of the current content (in this case the last part of the review which explains the book plot) which is irrelevant for our prediction.

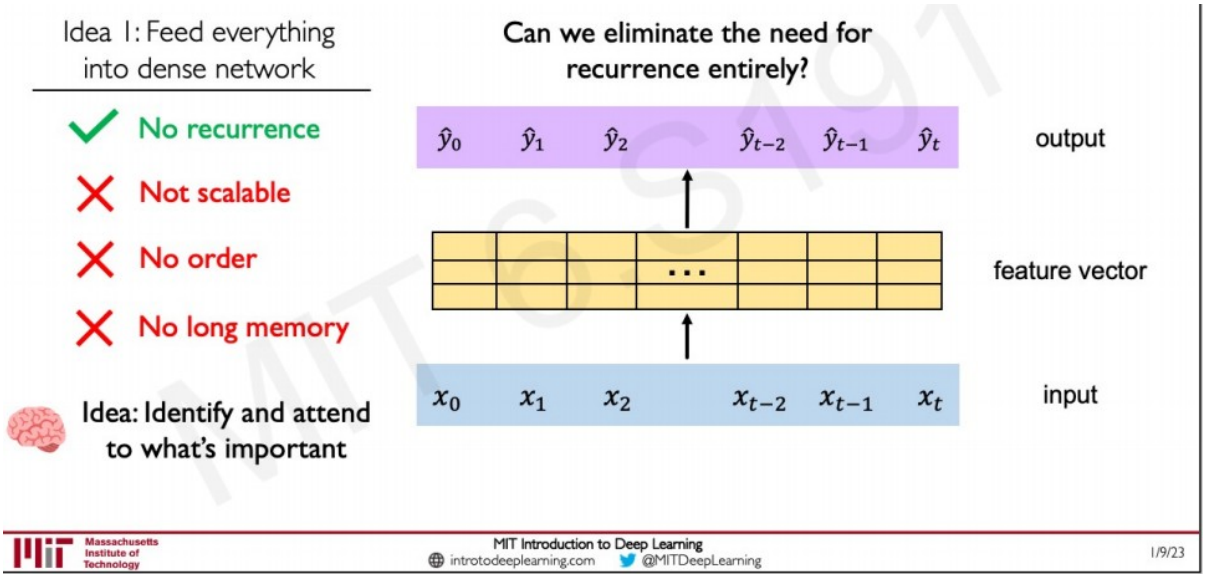
5.5.7. Limitations of RNNs and How to solve them?

1. **Encoding bottleneck:** For example we are trying to encode a lot of content, into a single output that may be at the very last time step, how do we ensure that all that information was maintained? In practice, this is very difficult and a lot of data is lost.
2. **Slow, no parallelization:** Doing this time step by time step computation, is very slow.
3. **Not long memory**: The biggest problem is that the capacity of RNNs and the LSTM is not that long, we can't handle big/massive data source effectively.

What we really want is:

1. **Continuous Stream**
2. **Parallelization**
3. **Long Memory**

One idea is to eliminate this need for recurrence entirely, maybe by squashing all these time steps together into a single feature vector, but as seen on the next page, this doesn't solve the issues:



Another *idea* is to identify and attend to what's important, that's where the idea of **Attention** comes from (see next section).

Hyperparameter/Layer type	What does it do?
Input text(s)	Target texts/sequences you'd like to discover patterns in
Input layer	Takes in target sequence
Text vectorization layer	Maps input sequences to numbers
Embedding	Turns mapping of text vectors to embedding matrix (representaiton of how words relate)
RNN cell(s)	Finds patterns in sequences
Hidden activation	Adds non-linearity to learned features (non-straight lines)
Pooling layer	Reduces the dimensionality of learned sequence features (usually for Conv1D models)
Fully connected layer	Further refines learned features from recurrent layers
Output layer	Takes learned features and outputs them in shape of target labels
Output activation	Adds non-linearities to output layer

Typical values
Whatever you can represent as text or a sequence
`input_shape = [batch_size, embedding_size]` or `[batch_size, sequence_shape]`
Multiple, can create with `tf.keras.layers.experimental.preprocessing.TextVectorization`
Multiple, can create with `tf.keras.layers.Embedding`
SimpleRNN, LSTM, GRU
Usually Tanh (hyperbolic tangent) (`tf.keras.activations.tanh`)
Average (`tf.keras.layers.GlobaelAveragePooling1D`) or Max (`tf.keras.layers.GlobalMaxPool1D`)
`tf.keras.layers.Dense`
`output_shape = [number_of_classes]` (e.g. 2 for Diaster, Not Diaster)
`tf.keras.activations.sigmoid` (binary classification) or `tf.keras.activations.softmax`

```
# 1. Create LSTM model
from tensorflow.keras import layers
inputs = layers.Input(shape=(1,), dtype="string")
x = text_vectorizer(inputs) # turn input sequence to numbers
x = embedding(x) # create embedding matrix
x = layers.LSTM(64, activation="tanh")(x) # return vector for whole sequence
outputs = layers.Dense(1, activation="sigmoid")(x) # create output layer
model = tf.keras.Model(inputs, outputs, name="LSTM_model")

# 2. Compile model
model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
              optimizer=tf.keras.optimizers.Adam(),
              metrics=["accuracy"])

# 3. Fit the model
history = model.fit(train_sentences, train_labels, epochs=5)
```

***There are almost an unlimited amount of ways to stack a RNN together, this is just one example.**

5.6. Model 2: LSTM (RNN)

To make sure we're not getting reusing trained embeddings **(this would involve data leakage between models, leading to an uneven comparison later on)**, we'll create another embedding layer (model_2_embedding) for our model. The *text_vectorizer layer* can be reused since it doesn't get updated during training.

The reason we use a new embedding layer for each model is since the embedding layer is a learned representation of words (as numbers), if we were to use the same embedding layer (embedding_1) for each model, we'd be mixing what one model learned with the next. And because we want to compare our models later on, starting them with their own embedding layer each time is a better idea.

```
# Set random seed and create embedding layer (new embedding layer for each model)
tf.random.set_seed(42)
from tensorflow.keras import layers
model_2_embedding = layers.Embedding(input_dim=max_vocab_length,
                                     output_dim=128,
                                     embeddings_initializer="uniform",
                                     input_length=max_length,
                                     name="embedding_2")


# Create LSTM model
inputs = layers.Input(shape=(1,), dtype="string")
x = text_vectorizer(inputs)
x = model_2_embedding(x)
print(x.shape)
# x = layers.LSTM(64, return_sequences=True)(x) # return vector for each word in the Tweet (you can stack RNN cells as long
x = layers.LSTM(64)(x) # return vector for whole sequence
print(x.shape)
# x = layers.Dense(64, activation="relu")(x) # optional dense layer on top of output of LSTM cell
outputs = layers.Dense(1, activation="sigmoid")(x)
model_2 = tf.keras.Model(inputs, outputs, name="model_2_LSTM")
```

```
(None, 15, 128)
(None, 64)
```

(Note on some hyperparameters: When we set `return_sequences=True`, the LSTM layer returns the full sequence of outputs, which means that the output of each timestep is returned. This is useful when we are stacking multiple LSTM layers on top of each other, as it ensures that the output of each layer has the same shape as the input

to the next layer.

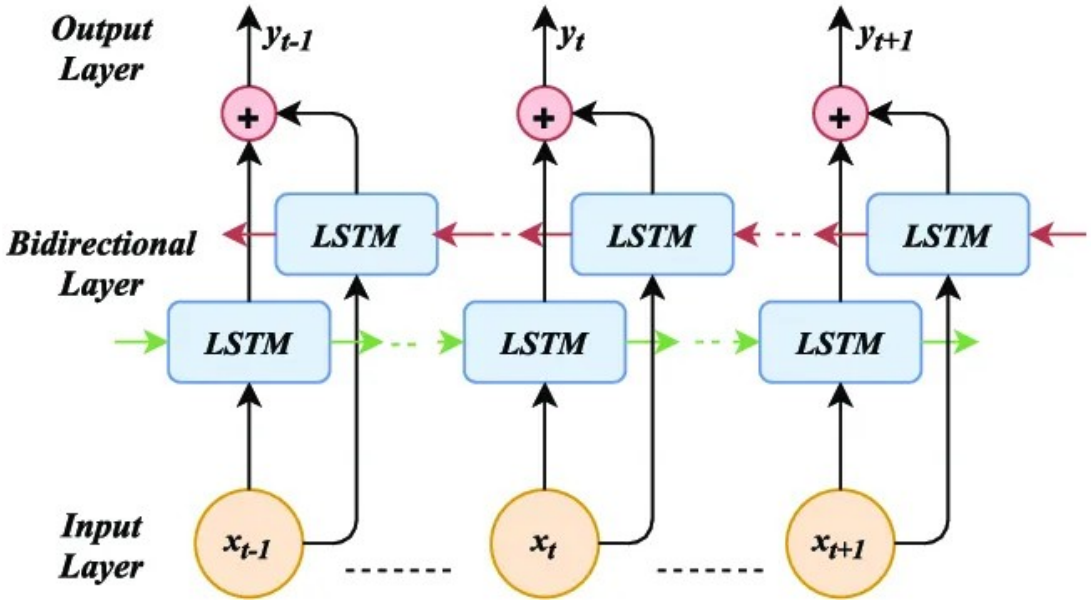
On the other hand, when we set `return_sequences=False` (which is the default value), the LSTM layer only returns the output of the last timestep. This is useful when we are using the LSTM layer as the final layer in our model, as we usually only need the final output of the LSTM layer.

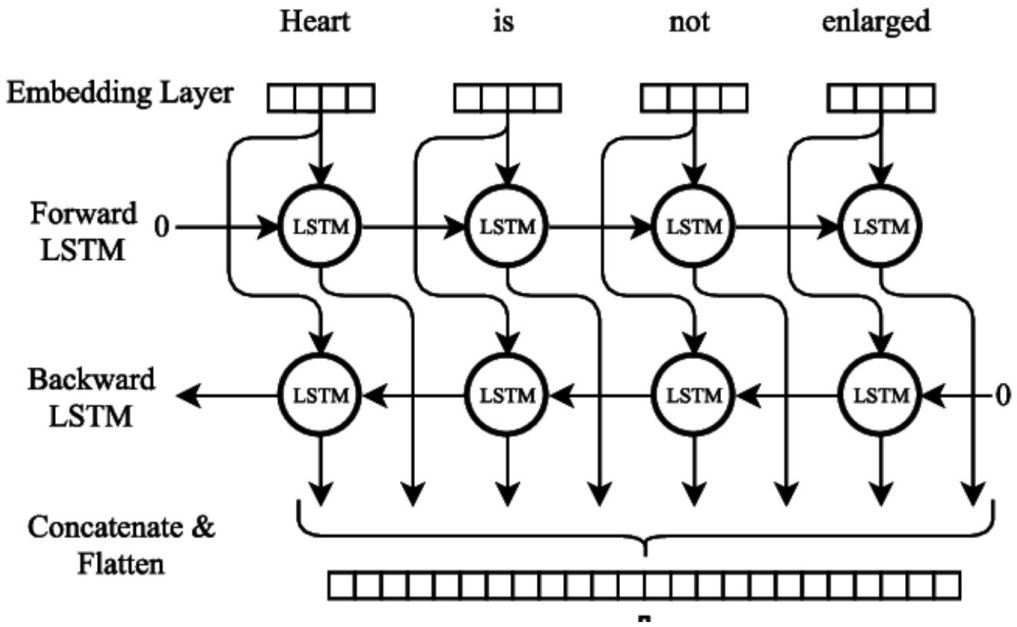
As for `return_state`, when we set it to `True`, the LSTM layer returns the final state of the layer in addition to the output. This is useful when we want to initialize the state of a subsequent LSTM layer with the final state of the previous layer).

5.7. Model 3: GRU (RNN)

```
# Set random seed and create embedding layer (new embedding layer for each model)
tf.random.set_seed(42)
from tensorflow.keras import layers
model_3_embedding = layers.Embedding(input_dim=max_vocab_length,
                                     output_dim=128,
                                     embeddings_initializer="uniform",
                                     input_length=max_length,
                                     name="embedding_3")

# Build an RNN using the GRU cell
inputs = layers.Input(shape=(1,), dtype="string")
x = text_vectorizer(inputs)
x = model_3_embedding(x)
# x = layers.GRU(64, return_sequences=True) # stacking recurrent cells requires return_sequences=True
x = layers.GRU(64)(x)
# x = layers.Dense(64, activation="relu")(x) # optional dense layer after GRU cell
outputs = layers.Dense(1, activation="sigmoid")(x)
model_3 = tf.keras.Model(inputs, outputs, name="model_3_GRU")
```

5.8. Model 4: Bidirectional LSTM (RNN)





Bi-LSTM (Bidirectional Long Short-Term Memory) is a type of RNN that processes sequential data in both forward and backward directions. It combines the power of LSTM with bidirectional processing, allowing the model to capture both past and future context of the input sequence.

Bi-LSTM processes the sequence in both directions simultaneously. **It consists of two LSTM layers: one processing the sequence in the forward direction and the other in the backward direction. Each layer maintains its own hidden states and memory cells.**

1. **Forward Pass:** *During the forward pass, the input sequence is fed into the forward LSTM layer from the first time step to the last.* At each time step, the forward LSTM computes its hidden state and updates its memory cell based on the current input and the previous hidden state and memory cell.
2. **Backward Pass:** *Simultaneously, the input sequence is also fed into the backward LSTM layer in reverse order, from the last time step to the first*. Similar to the forward pass, the backward LSTM computes its hidden state and updates its memory cell based on the current input and the previous hidden state and memory cell.
3. **Combining Forward and Backward States:** *Once the forward and backward passes are complete, the hidden states from both LSTM layers are combined at each time step. This combination can be as simple as concatenating the hidden states or applying some other transformation.*

Pros and Cons:

<u>Pros:</u>

1. **Captures long-term dependencies:** Bidirectional LSTMs can capture long-term dependencies in sequential data by processing input sequences in both forward and backward directions. This makes them well-suited for tasks that require modeling context over a long period of time, such as speech recognition or natural language processing.

2. **Improved performance:** Bidirectional LSTMs often perform better than traditional LSTMs on tasks such as speech recognition, machine translation, and sentiment analysis.

3. **Flexible architecture:** The architecture of bidirectional LSTMs is flexible and can be customized by adding additional layers, such as convolutional or attention layers, to improve performance.

Cons:

1. **High computational cost:** The improvement in performance often comes at the cost of longer training times and increased model parameters (since the model goes left to right and right to left, the number of trainable parameters doubles). This can make them impractical for use in resource-constrained environments.

2. **Requires large amounts of data:** Bidirectional LSTMs require large amounts of training data to learn meaningful representations of the input sequence. Without sufficient training data, the model may overfit to the training set or fail to generalize to new data.

3. **Difficult to interpret:** Bidirectional LSTMs are often seen as “black boxes,” making it difficult to interpret how the model is making predictions. This can be problematic in applications where interpretability is important,

5.9. CNNs for Sequence Problems

(Explanations in this section are based on paper: [1809.08037] Understanding Convolutional Neural Networks for Text Classification (arxiv.org), read it for details and proof)

CNNs, originally invented for computer vision, have been shown to achieve strong performance on text classification tasks as well as other traditional NLP tasks, even when considering relatively simple one-layer models.

Current common wisdom suggests that CNNs classify text by working through the following steps:

1. 1-dimensional convolving filters are used as **ngram detectors**, each filter specializing in a closely-related family of ngrams.

   *(Note: Remember from **IntroductionAI** that an n-gram is a sequence of n items from a sample of text. In a character n-gram, the items are characters, and in a word n-gram the items are words. A unigram, bigram, and trigram are sequences of one, two, and three items)*

2. Max-pooling over time extracts the relevant ngrams for making a decision.

3. The rest of the network classifies the text based on this information.

In the paper the authors refine items 1 and 2 and show that:

- *Max-pooling induces a thresholding behavior, and values below a given threshold are ignored when (i.e. irrelevant to) making a prediction*. Specifically, and show an experiment for which 40% of the pooled ngrams on average can be dropped with no loss of performance (Section 4).

- *Filters are not homogeneous* (i.e. a single filter can, and often does, detect multiple distinctly different families of ngrams)

- *Filters also detect negative items in ngrams— they not only select for a family of ngrams but often actively suppress a related family of negated ngrams.*

5.9.1. Inner workings

CNN filters serve as **ngram detectors**: each filter searches for a specific class of ngrams, which it marks by assigning them high scores. These highest-scoring detected ngrams survive the max-pooling operation.

The final decision is then based on the set of ngrams in the max-pooled vector (represented by the set of corresponding filters). Intuitively, ngrams which any filter scores highly (relative to how it scores other ngrams) are ngrams which are highly relevant for the classification of the text.

5.10. Model 5: Conv1D

```
# Test out our embedding layer, Conv1D layer and max pooling
from tensorflow.keras import layers
embedding_test = embedding(text_vectorizer(["this is a test sentence"])) # turn target sequence into embedding
conv_1d = layers.Conv1D(filters=32,
                        kernel_size=5, # this is also referred to as an ngram of 5 (meaning it looks at 5 words at a time)
                        strides=1, # default
                        activation="relu",
                        padding="same") # default = "valid", the output is smaller than the input shape, "same" means output is
conv_1d_output = conv_1d(embedding_test) # pass test embedding through conv1d layer
max_pool = layers.GlobalMaxPool1D()
max_pool_output = max_pool(conv_1d_output) # equivalent to "get the most important feature" or "get the feature with the highes

embedding_test.shape, conv_1d_output.shape, max_pool_output.shape
```

```
(TensorShape([1, 15, 128]), TensorShape([1, 15, 32]), TensorShape([1, 32]))
```

What padding=”same” will do is that it will add zeros if it needs to to preserve all the data in the sequence:

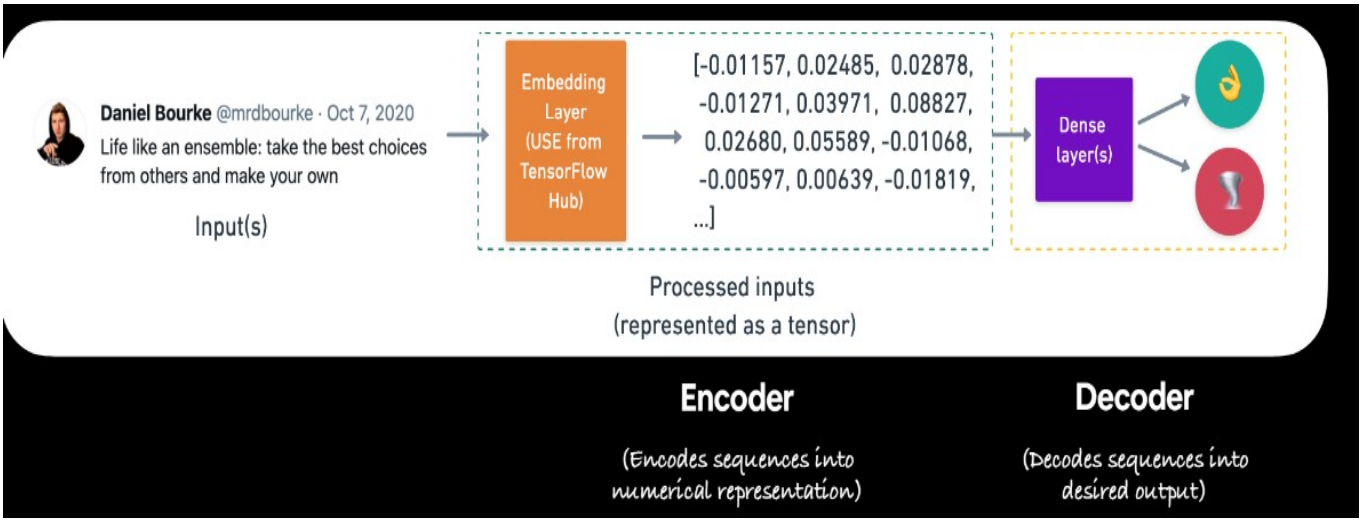
- `"VALID"` = without padding:

```
inputs:         1  2  3  4  5  6  7  8  9  10 11 (12 13)
               |_________________|                dropped
                     |_________________|
```

- `"SAME"` = with zero padding:

```
             pad|                                        |pad
inputs:       0 |1  2  3  4  5  6  7  8  9  10 11 12 13|0  0
              |_________________|
                    |_________________|
                          |_________________|
```

5.11. Model 6: Using Transfer Learning (Pretrained Sentence Encoder)



In this case we'll use the **Universal Sentence Encoder (USE)**, we've been using word embeddings for our models, **USE** does a sentence embedding. (Note: Another possible choice could be **Bidirectional Encoder Representations from Transformers (BERT)**. But I haven't talked about transformers yet)

From the paper: *"… encoding sentences into embedding vectors that specifically target transfer learning to other NLP tasks. The models are efficient and result in accurate performance on diverse transfer tasks. Two variants of the encoding models allow for trade-offs between accuracy and compute resources …*

… The model is trained and optimized for greater-than-word length text, such as sentences, phrases or short paragraphs. It is trained on a variety of data sources and a variety of tasks with the aim of dynamically accommodating a wide variety of natural language understanding tasks. The input is variable length English text and the output is a 512 dimensional vector …"

We can convert the TensorFlow Hub USE module into a Keras layer using the hub.KerasLayer class.

```
# We can use this encoding layer in place of our text_vectorizer and embedding layer
sentence_encoder_layer = hub.KerasLayer("https://tfhub.dev/google/universal-sentence-encoder/4",
                                        input_shape=[], # shape of inputs coming to our model
                                        dtype=tf.string, # data type of inputs coming to the USE layer
                                        trainable=False, # keep the pretrained weights (we'll create a feature extractor)
                                        name="USE")
```

```
# Create model using the Sequential API
model_6 = tf.keras.Sequential([
  sentence_encoder_layer, # take in sentences and then encode them into an embedding
  layers.Dense(64, activation="relu"),
  layers.Dense(1, activation="sigmoid")
], name="model_6_USE")

# Compile model
model_6.compile(loss="binary_crossentropy",
                optimizer=tf.keras.optimizers.Adam(),
                metrics=["accuracy"])

model_6.summary()
```

```
Model: "model_6_USE"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 USE (KerasLayer)            (None, 512)               256797824

 dense_5 (Dense)             (None, 64)                32832

 dense_6 (Dense)             (None, 1)                 65

=================================================================
Total params: 256,830,721
Trainable params: 32,897
Non-trainable params: 256,797,824
_________________________________________________________________
```

The trainable parameters are only in our output layers, in other words, we're keeping the USE weights frozen and using it as a feature-extractor. We could fine-tune these weights by setting `trainable=True` when creating the `hub.KerasLayer` instance.

5.12. Model 7: Same as model 6 but with 10% of the data

In practice (e.g. our job), we may not have as much data as in a complete dataset, and we may have to work with less, that's the purpose of this model:

```
### NOTE: Making splits like this will lead to data leakage ###
### (some of the training examples in the validation set) ###

### WRONG WAY TO MAKE SPLITS (train_df_shuffled has already been split) ###

# # Create subsets of 10% of the training data
# train_10_percent = train_df_shuffled[["text", "target"]].sample(frac=0.1, random_state=42)
# train_sentences_10_percent = train_10_percent["text"].to_list()
# train_labels_10_percent = train_10_percent["target"].to_list()
# len(train_sentences_10_percent), len(train_labels_10_percent)
```

```
# One kind of correct way (there are more) to make data subset
# (split the already split train_sentences/train_labels)
train_sentences_90_percent, train_sentences_10_percent, train_labels_90_percent, train_labels_10_percent = train_test_split(
```

```
# Check length of 10 percent datasets
print(f"Total training examples: {len(train_sentences)}")
print(f"Length of 10% training examples: {len(train_sentences_10_percent)}")
```

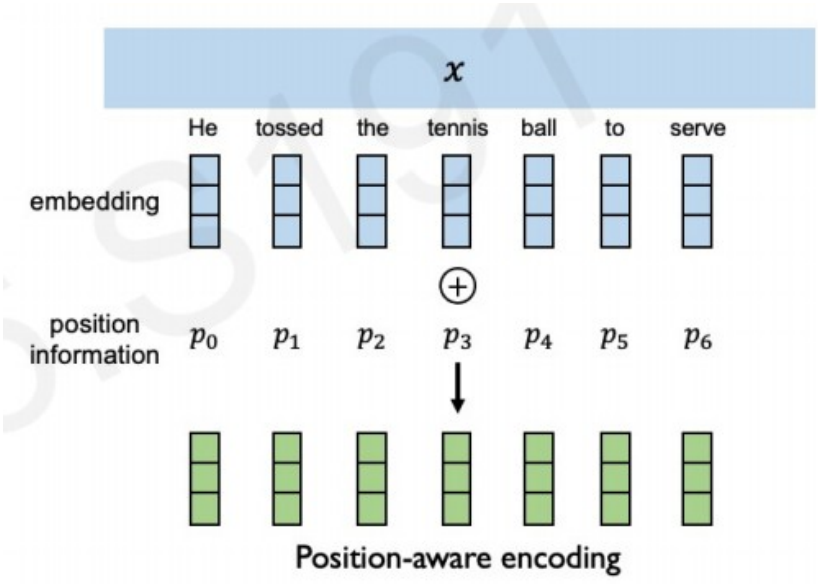
5.13. Attention

One of the ways to solve the limitations of RNNs (*see section 5.5.7*) is with an extremely powerful concept called **attention**. **The key idea of attention is to eliminate recurrence and attend to the most important features of the information**. **Attention is the key idea behind Transformers.**

For example: When we search for a video in youtube we have a **query** (*Q*) and when we search we get some **keys** (*K)* that ideally have some similarity with our search query, what we need is a *metric of similarity between our query and these keys* and the **value** is the actual video that has the *highest attention*, the most relevant video.

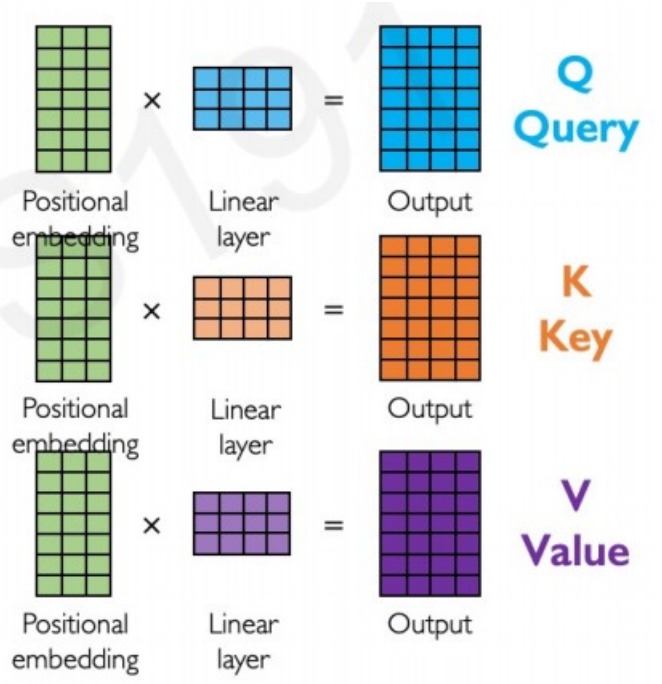
Let's dive into the inner workings of attention step by step**, our goal is to identify and attend to the most important features of our input**:

1. Encode **position** information:



In order to do this data is fed all in at once! We need to encode position information to understand order.

2. Extract **query, key, value** for search

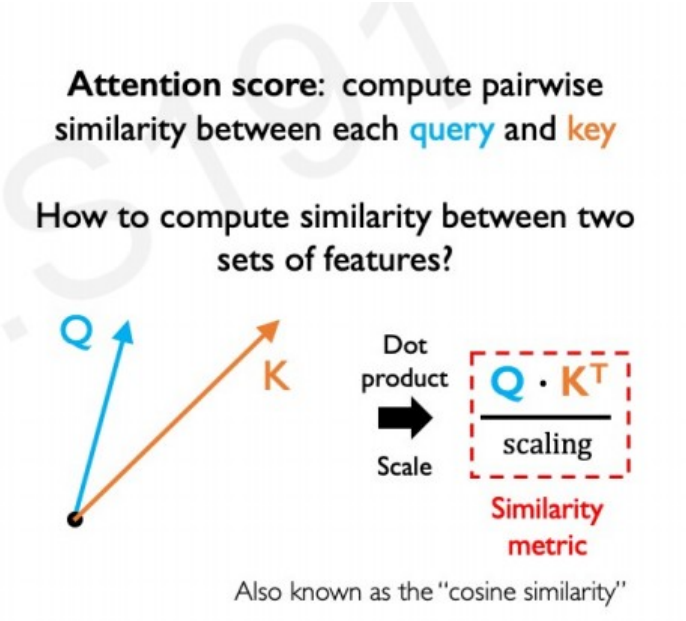


Given the positional embedding calculated earlier, we apply a linear layer generating the **query**. We do this again using another separate layer (different set of weights) transforming the positional embedding in a different way, generating a **key**. This is repeated again to generate a **value**.

What we now need to do is to compare these three to try to figure out where in that input, the network should pay attention to.

3. Compute **attention weighting**

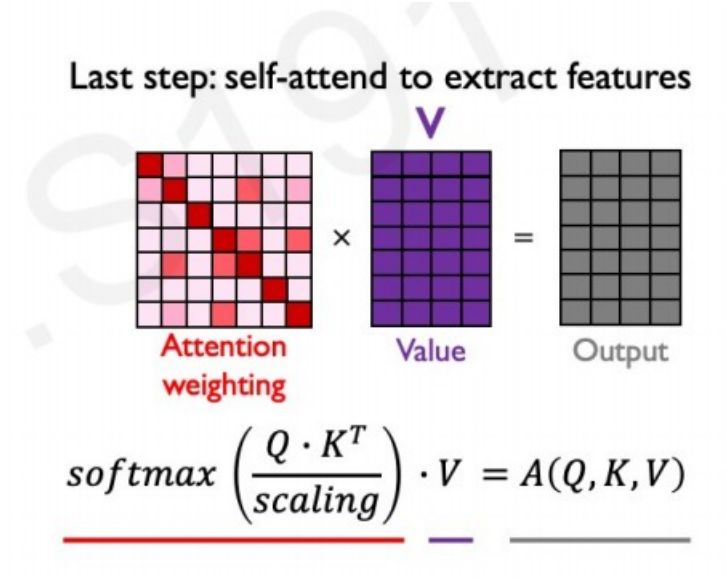
   The tool that allows us to compare the similarity is the *dot product and scaling it:*



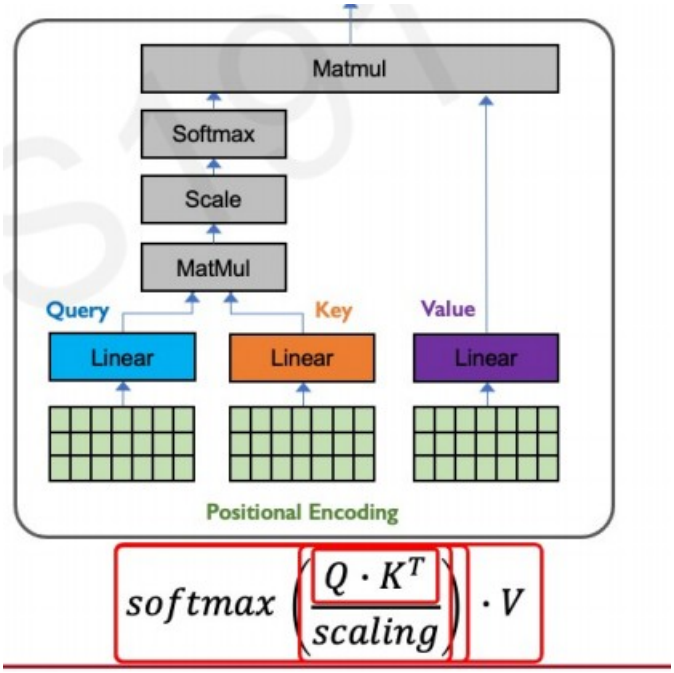
   We then apply a **softmax** to this attention score, to set the values between 0 and 1.

   (More about "scaling" in the transformers section)

4. Extract **features with high attention**



So, in summary, this is what a **single attention head looks like:**



Multiple of these can be linked together to build larger models. The adding of multiple heads results in a Transformer Architecture, more on this later.

Please, have in mind that this is the explanation for "**self-attention**" which is not the same as **"vanilla attention".**

Before the introduction of the Transformer model, attention mechanisms were implemented by RNN-based encoder-decoder architectures. The name comes from the fact that contrary to "regular" attention, **self-attention refers to the the same sequence which is currently being encoded.**

5.13.1. Vanilla Attention

Before the introduction of the Transformer model, attention mechanisms were primarily used in conjunction with Recurrent Neural Network (RNN)-based encoder-decoder architectures for tasks such as neural machine translation. Two of the most popular models that implemented attention in this manner were those proposed by Bahdanau et al. (2014) and Luong et al. (2015).

You can find an example here: (Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention) – Jay Alammar – Visualizing machine learning one concept at a time. (jalammar.github.io))

These models consist of an **encoder and decoder**. The *encoder* processes each item in the input sequence, it compiles the information it captures into a vector (called the context). After processing the entire input sequence, the encoder sends the context over to *the decoder*, which begins producing the output sequence item by item.

These models are also almost always based on Recurrent neural networks.

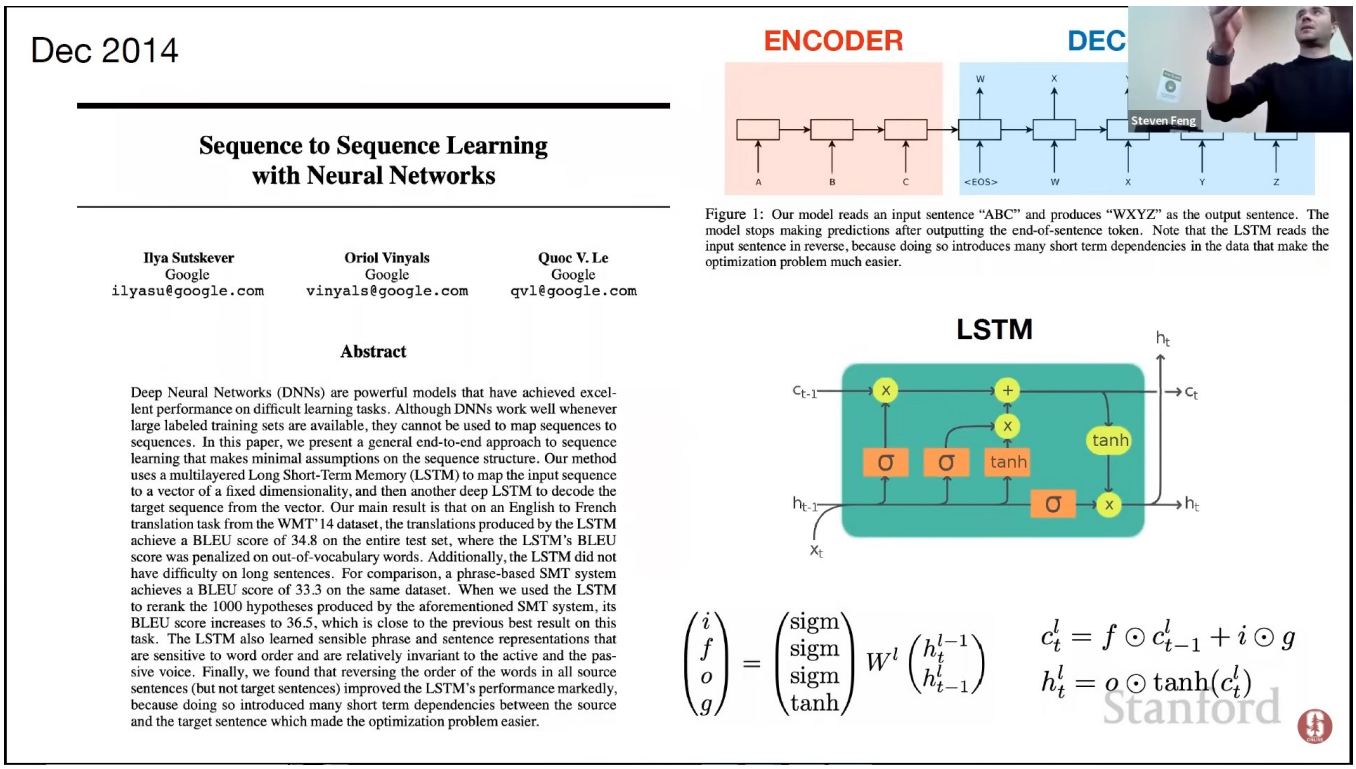
5.14. Transformers

(Note: The information here is mainly derived from the paper: *Attention is All You* Need. Before starting this chapter I want to highly encourage the reader to actually read the paper, it's very comprehensible once you understand all the previous subjects. This section is meant to be read as complementary to it)

5.14.1. History

(Note: Information extracted mainly from: [Stanford CS25: V2 I Introduction to Transformers w/ Andrej Karpathy (youtube.com)])

Around December 2014 the goal was to produce effective machine translation (go from an english sentence to a french sentence), in this paper they basically have an encoder LSTM in the left, "consumes" every word to be translated at a time and builds a context vector of what it has read, then that's passed to the decoder that goes trying to predict the next word in french.



The problem with this is that there's what called the "*encoder bottleneck*" problem. The entire english sentence is packed into a single vector and that's too much information to have in a single vector, which causes problems.

The way to solve this is to use **attention,** this paper below is the first introduction of attention, this was a way to look back at the words coming from the encoder, it was done using this *soft-search*, as you are decoding the words you're allowed to look back at the words of the encoder.

Sep 2014

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

ABSTRACT

Neural machine translation is a recently proposed approach to machine translation. Unlike the traditional statistical machine translation, the neural machine translation aims at building a single neural network that can be jointly tuned to maximize the translation performance. The models proposed recently for neural machine translation often belong to a family of encoder–decoders and encode a source sentence into a fixed-length vector from which a decoder generates a translation. In this paper, we conjecture that the use of a fixed-length vector is a bottleneck in improving the performance of this basic encoder–decoder architecture, and propose to extend this by allowing a model to automatically (soft-)search for parts of a source sentence that are relevant to predicting a target word, without having to form these parts as a hard segment explicitly. With this new approach, we achieve a translation performance comparable to the existing state-of-the-art phrase-based system on the task of English-to-French translation. Furthermore, qualitative analysis reveals that the (soft-)alignments found by the model agree well with our intuition.

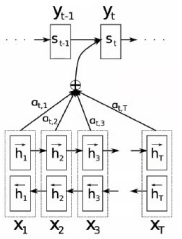
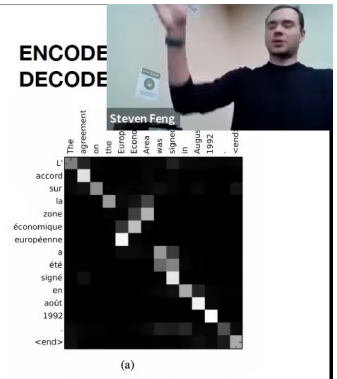


Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.



(a)

The context vector c_i is, then, computed as a weighted sum of these annotations h_i:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j. \quad (5)$$

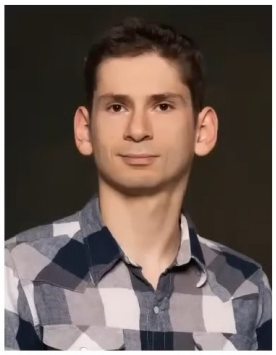
The weight α_{ij} of each annotation h_j is computed by

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})},$$

where

$$e_{ij} = a(s_{i-1}, h_j)$$

respect to the previous hidden state s_{i-1} in deciding the next state s_i and generating y_i. Intuitively, this implements a mechanism of attention in the decoder. The decoder decides parts of the source sentence to pay attention to. By letting the decoder have an attention mechanism, we relieve the encoder from the burden of having to encode all information in the source sentence into a fixed-length vector. With this new approach the information can be spread throughout the sequence



Dzmitry Bahdanau

RE: history of "attention"
(from email correspondence)

So I started thinking about how to avoid the bottleneck between encoder and decoder RNN. My first idea was to have a model with two "cursors", one moving through the source sequence (encoded by a BiRNN) and another one moving through the target sequence. The cursor trajectories would be marginalized out using dynamic programming. KyungHyun Cho recognized this as an equivalent to Alex Graves' RNN Transducer model. Following that, I may have also read Graves' hand-writing recognition paper. The approach looked inappropriate for machine translation though.

The above approach with cursors would be too hard to implement in the remaining 5 weeks of my internship. So I tried instead something simpler - two cursors moving at the same time synchronously (effectively hard-coded diagonal attention). That sort of worked, but the approach lacked elegance.

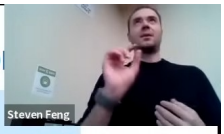
So one day I had this thought that it would be nice to enable the decoder RNN to learn to search where to put the cursor in the source sequence. This was sort of inspired by translation exercises that learning English in my middle school involved. Your gaze shifts back and forth between source and target sequence as you translate. I expressed the soft search as softmax and then weighted averaging of BiRNN states. It worked great from the very first try to my great excitement. I called the architecture RNNSearch, and we rushed to publish an ArXiV paper as we knew that Ilya and co at Google are somewhat ahead of us with their giant 8 GPU LSTM model (RNN Search still ran on 1 GPU).

As it later turned out, the name was not great. The better name (attention) was only added by Yoshua to the conclusion in one of the final passes.

Stanford

This bring us to the transformer architecture proposed in 2017, that introduced the idea that attention was all that was needed and we could just drop everything else:

Dec 2017



Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

"Full package" model:
- Everything only Attention, delete all RNN components
- Positional encodings
- Residual network (ResNet) structure
- Interspersing of Attention and MLP
- LayerNorms
- Multiple heads of attention in parallel
- Great hyperparameters (e.g. ffw_size=4, isotropic)
- …

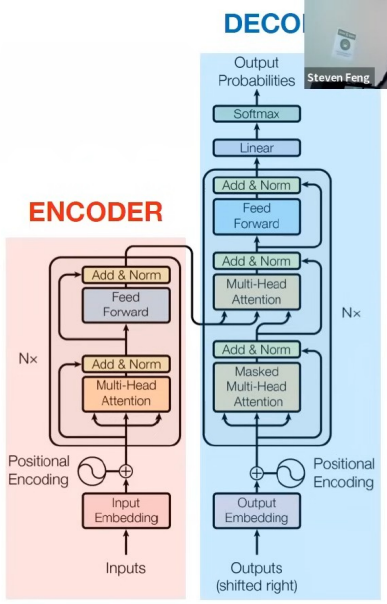


Figure 1: The Transformer - model architecture.

The transformer has changed remarkably little to this day.
The only consistent change is the "pre-norm" formulation, reshuffling the LayerNorms

Stanford

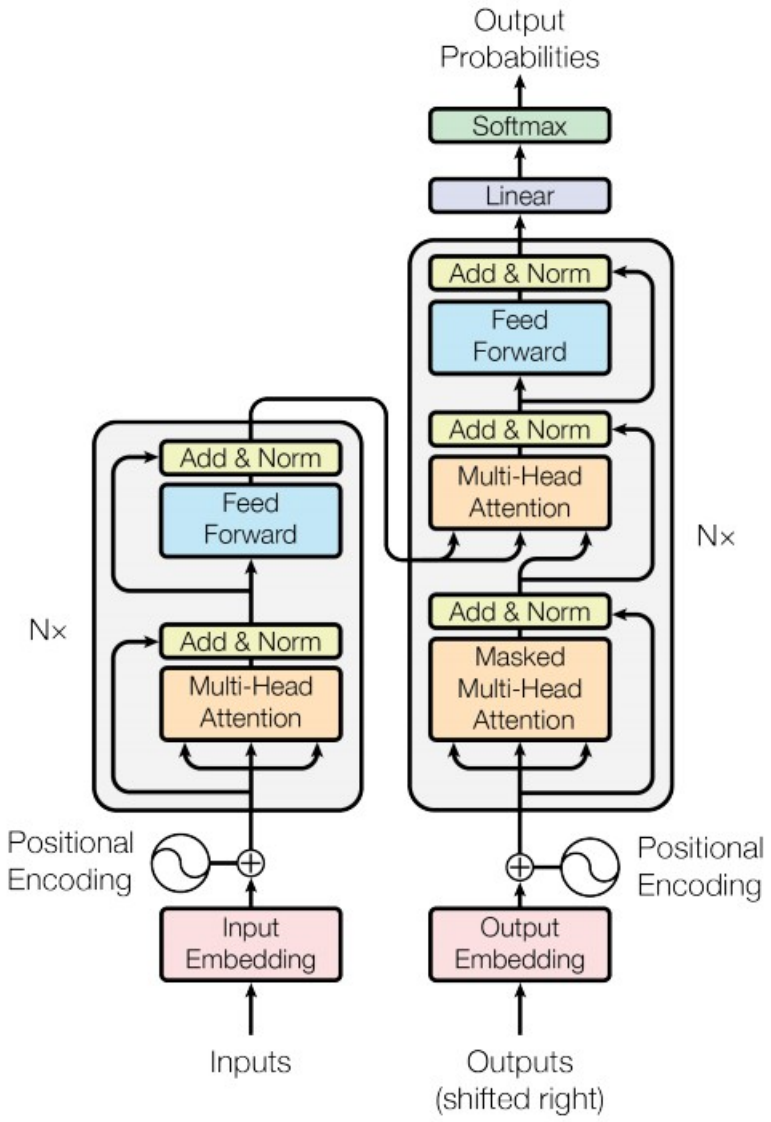


Figure 1: The Transformer - model architecture.

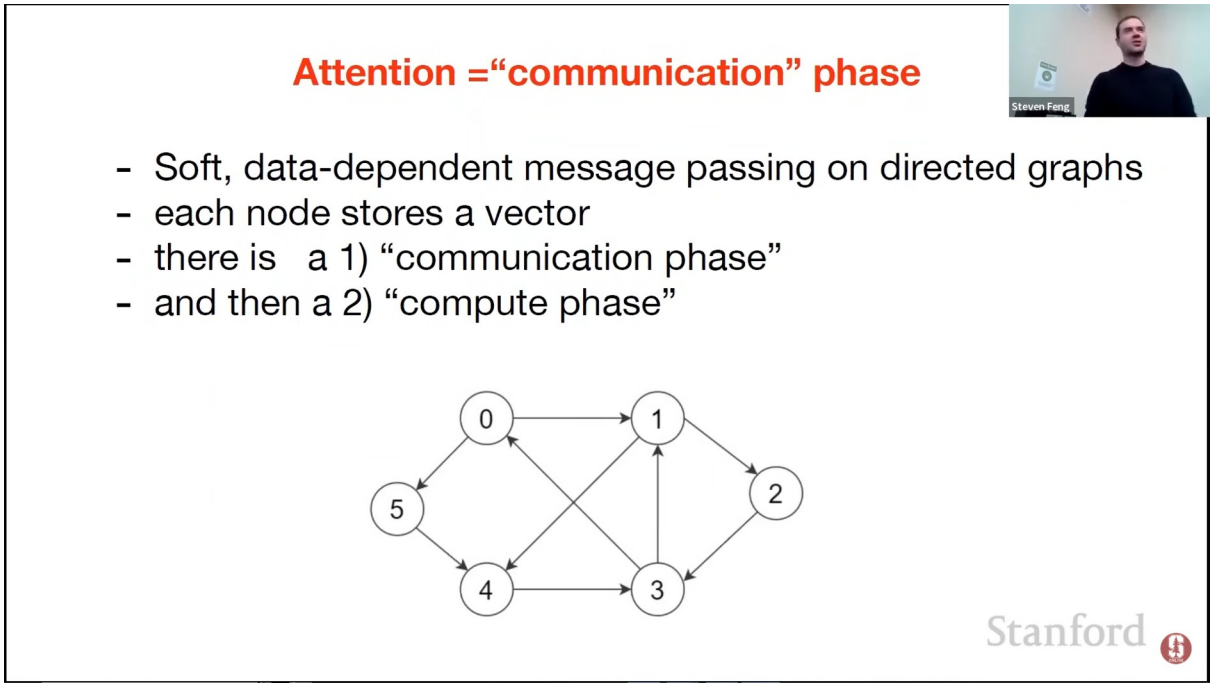
5.14.2. Multi-Head Attention

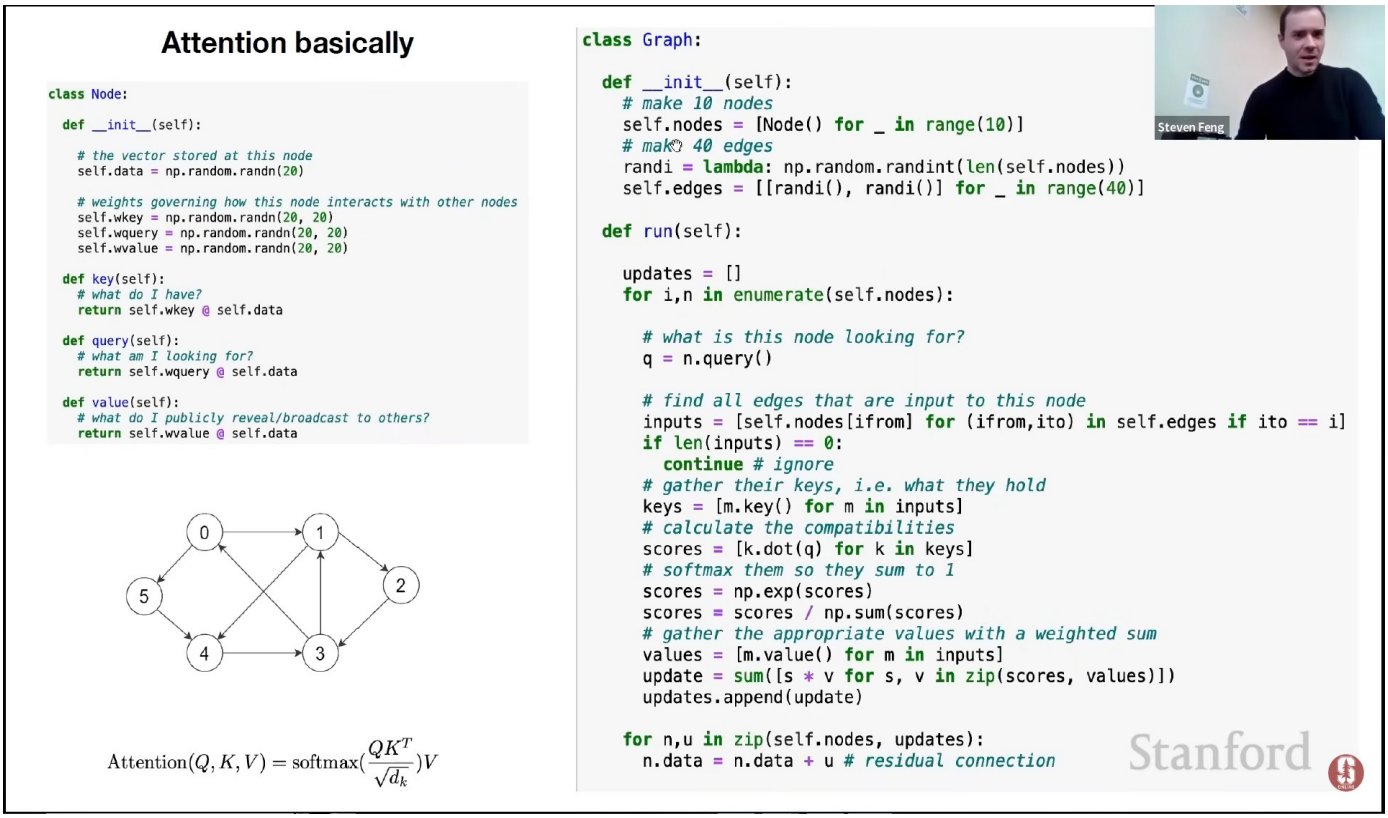
In the section: **3.2.1 Scaled Dot-Product Attention** of the paper we get some clearance as to what *"scaling"* means (as introduced in the previous chapter), we are given the following formula:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Where $\sqrt{d_k}$ is the square root of the dimension of the key vector. This is done in order to prevent the dot product from becoming too large or too small.

In Andrej's lecture we see that attention can be thought of as kind of a directed graph, where each node would represent a vector (i.e. each embedded word), **the communication phase** (multi-headed attention) **the compute phase** is just the multi-layer perceptron, which will act on every node individually.





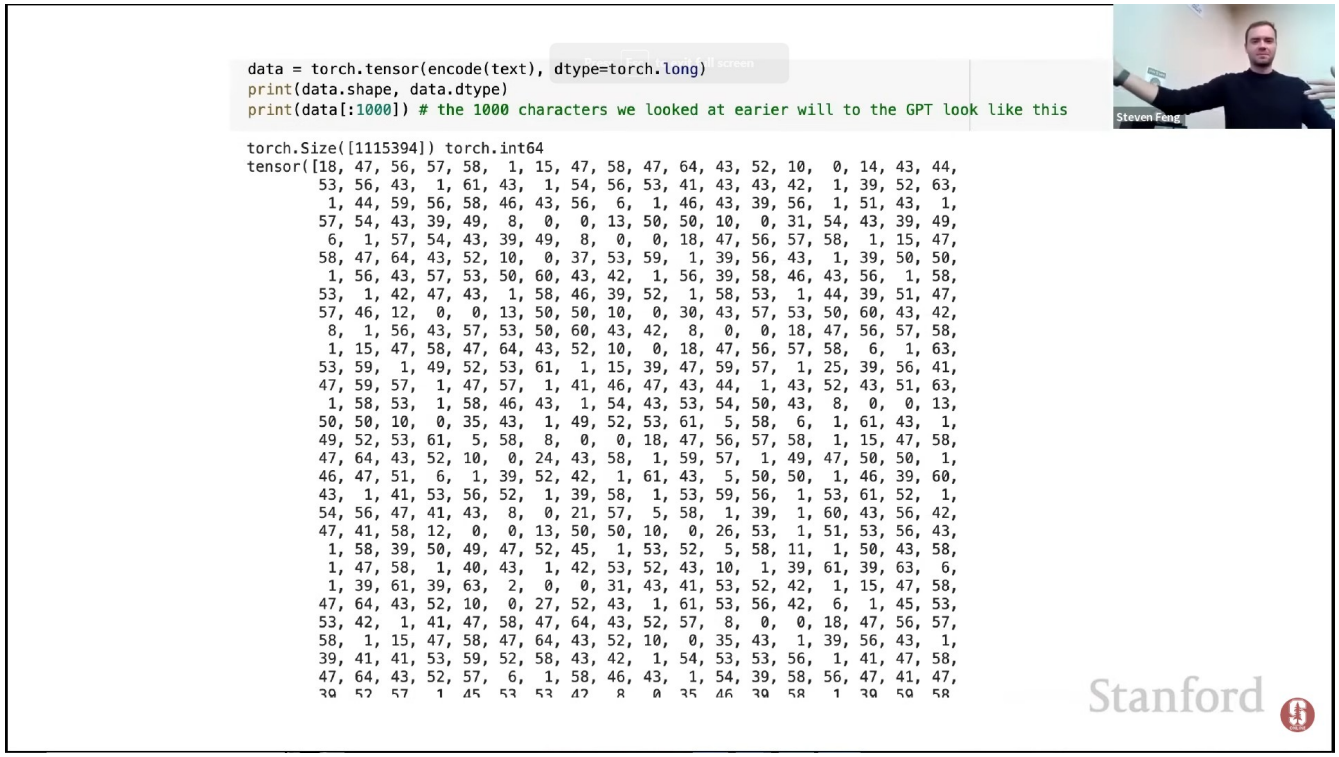
(Note: this is a simplification, some other things happen such as layer normalization)

This attention scheme happens in every head in parallel and then in every layer, in series.

Notice that this particular attention explained is **self-attention**, this is because every one of those vectors produces a *key, query and value* from this individual vector, in **cross-attention** (you have one in the decoder, coming from the encoder) that just means that the *queries* are still produced from the vector itself, but the keys and the values are produced as a function of the vectors coming from the encoder.

Example: I have my queries because I'm trying to decode the fifth word in a given sequence, so the keys and the values (the source of information that can answer my queries) can come from the previous nodes in the current decoding sequence OR from the top of the encoding sequence.

5.14.3. Transformer-based GPT example



We encode every single character as an integer and we concatenate it into one tensor as you see above. In this example the training is done with a single Shakespeare document but if we had many what's often done is to **create special tokens** (End of text tokens) that indicate where a document ends, creating boundaries between documents.

These boundaries don't have any modeling impact, but the transformer is supposed to learn that the end of document token is supposed to indicate to wipe out the memory.

The next thing we do is to produce **batches of data** (We take chunks of the original sequence), this also let's us know how many sequences in parallel we're going to process, helping us speed up the process. The **block size** indicates the maximum length of context that the transformer will process, in this case below we have 8 meaning that we'll have up to 8 characters of context to predict the 9th character in the sequence.

Build a batch of data

Each batch of data is an (x,y) tuple
- x is the input
- y is the desired output

Both x,y are BxT integer tensor


```
[ ] batch_size = 4
    block_size = 8
    def get_batch(split):
        data = train_data if split == 'train' else val_data
        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
```

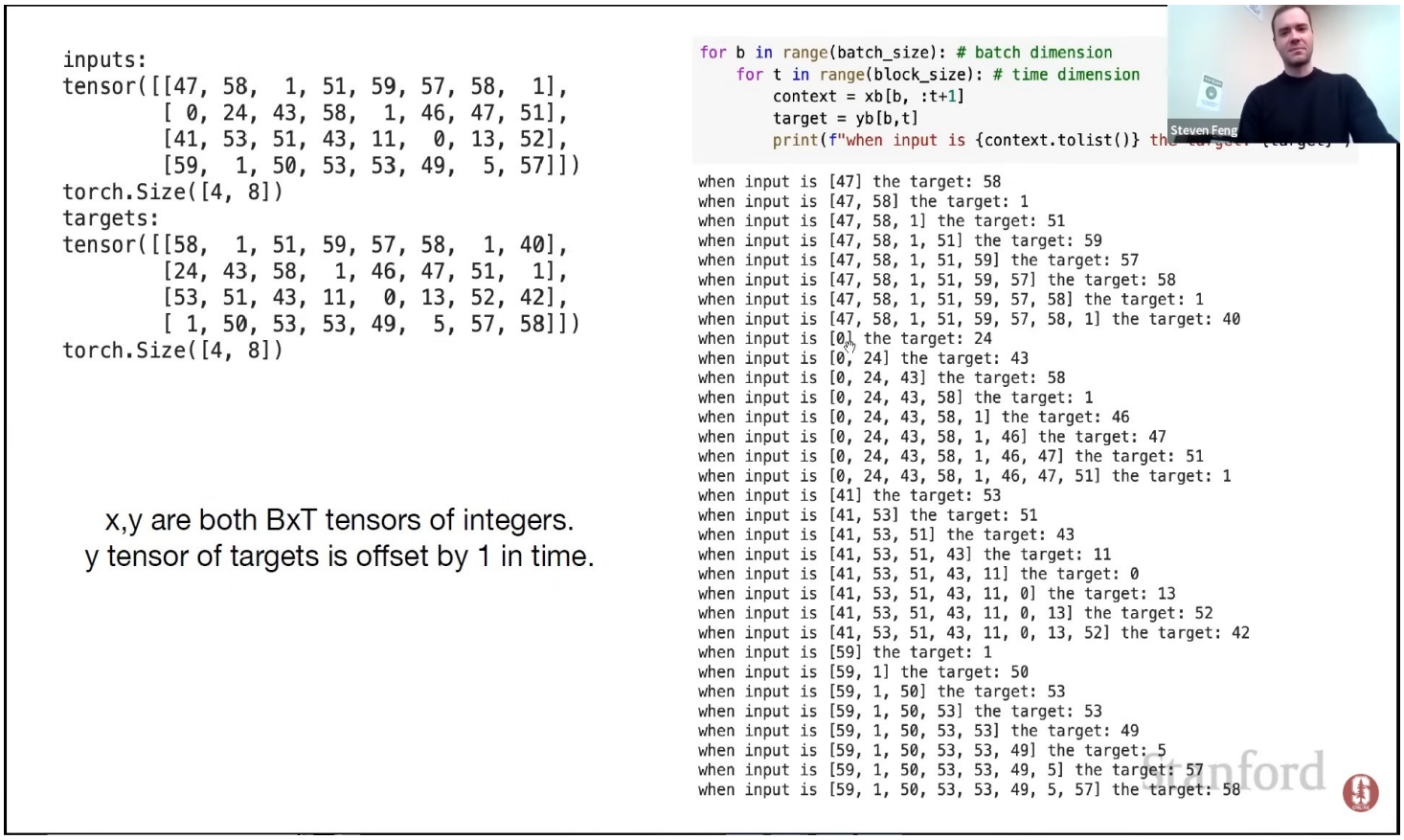
In this chunk of code:

- `batch_size` is how many independent chunks of text we have in the batch
- `block_size` is the length of each sequence chunk
- `x` is going to be the input to the GPT
- `y` will be the desired output from the GPT

```
[ ] xb, yb = get_batch('train')
    print('inputs:')
    print(xb)
    print(xb.shape)
    print('targets:')
    print(yb)
    print(yb.shape)
```

```
inputs:
tensor([[47, 58,  1, 51, 59, 57, 58,  1],
        [ 0, 24, 43, 58,  1, 46, 47, 51],
        [41, 53, 51, 43, 11,  0, 13, 52],
        [59,  1, 50, 53, 53, 49,  5, 57]])
torch.Size([4, 8])
targets:
tensor([[58,  1, 51, 59, 57, 58,  1, 40],
        [24, 43, 58,  1, 46, 47, 51,  1],
        [53, 51, 43, 11,  0, 13, 52, 42],
        [ 1, 50, 53, 53, 49,  5, 57, 58]])
torch.Size([4, 8])
```

Stanford



(Note: Andrej's words: "The batches are all learned all completely independently but the **time dimension** here ("time dimension" refers to the sequence length or the number of time steps in a sequence (e.g., words in a sentence)) horizontally is also trained in parallel (remember that in a transformer, the time dimension is treated differently. Instead of processing one time step at a time, transformers parallelize the training process)

so the real batch size is more like B * T, the context grows linearly for the predictions that you make along the T direction.")

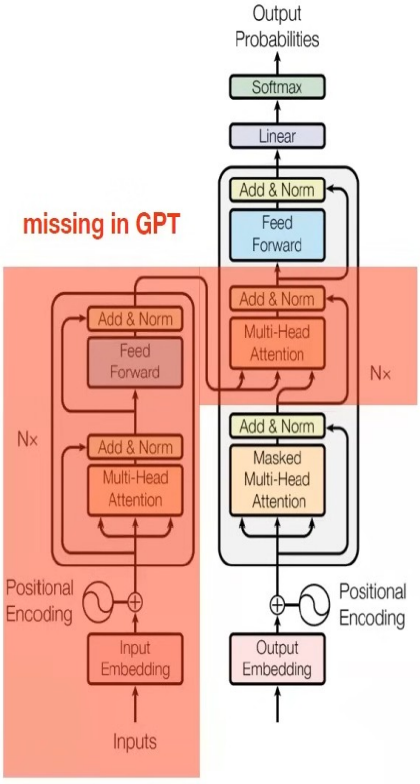


Figure 1: The Transformer - model architecture.

```python
class GPT(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.vocab_size is not None
        assert config.block_size is not None
        self.config = config

        self.transformer = nn.ModuleDict(dict(
            wte = nn.Embedding(config.vocab_size, config.n_embd),
            wpe = nn.Embedding(config.block_size, config.n_embd),
            drop = nn.Dropout(config.dropout),
            h = nn.ModuleList([Block(config) for _ in range(config.n_layer)]),
            ln_f = nn.LayerNorm(config.n_embd),
        ))
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)

    def forward(self, idx, targets=None):
        device = idx.device
        b, t = idx.size()
        assert t <= self.config.block_size, f"Cannot forward sequence of length {t}, block size is only
        pos = torch.arange(0, t, dtype=torch.long, device=device).unsqueeze(0) # shape (1, t)

        # forward the GPT model itself
        tok_emb = self.transformer.wte(idx) # token embeddings of shape (b, t, n_embd)
        pos_emb = self.transformer.wpe(pos) # position embeddings of shape (1, t, n_embd)
        x = self.transformer.drop(tok_emb + pos_emb)
        for block in self.transformer.h:
            x = block(x)
        x = self.transformer.ln_f(x)
        logits = self.lm_head(x)

        # if we are given some desired targets also calculate the loss
        loss = None
        if targets is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1), ignore_index=-1)

        return logits, loss
```

```
inputs:
tensor([[
torch.Siz
targets:
tensor([[
        [53, 51, 43, 11,  0, 13, 52, 42],
        [ 1, 50, 53, 53, 49,  5, 57, 58]])
torch.Size([4, 8])
```

idx, targets
i.e. x,y

output logits are of shape (B,T,vocab_size)

Stanford

Notice that this is a decoder-only model, this is because we're not trying to condition on some other external information, we're just trying to produce a sequence of words that are likely to follow each other.

```
class GPT(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.vocab_size is not None
        assert config.block_size is not None
        self.config = config

        self.transformer = nn.ModuleDict(dict(
            wte = nn.Embedding(config.vocab_size, config.n_embd),
            wpe = nn.Embedding(config.block_size, config.n_embd),
            drop = nn.Dropout(config.dropout),
            h = nn.ModuleList([Block(config) for _ in range(config.n_layer)]),
            ln_f = nn.LayerNorm(config.n_embd),
        ))
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)

    def forward(self, idx, targets=None):
        device = idx.device
        b, t = idx.size()
        assert t <= self.config.block_size, f"Cannot forward sequence of length {t}, block size is only
        pos = torch.arange(0, t, dtype=torch.long, device=device).unsqueeze(0) # shape (1, t)

        # forward the GPT model itself
        tok_emb = self.transformer.wte(idx) # token embeddings of shape (b, t, n_embd)
        pos_emb = self.transformer.wpe(pos) # position embeddings of shape (1, t, n_embd)
        x = self.transformer.drop(tok_emb + pos_emb)
        for block in self.transformer.h:
            x = block(x)
        x = self.transformer.ln_f(x)
        logits = self.lm_head(x)

        # if we are given some desired targets also calculate the loss
        loss = None
        if targets is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1), ignore_index=-1)

        return logits, loss
```

```
inputs:
tensor([[
        [
        [
        [
torch.Siz
targets:
tensor([[
        [
        [53, 51, 43, 11,  0, 13, 52, 42],
        [ 1, 50, 53, 53, 49,  5, 57, 58]])
torch.Size([4, 8])
```

idx, targets i.e. x,y

output logits are of shape (B,T,vocab_size)

In the forward pass we take these indices and then we encode the identity of the indices via an embedding lookup table. **Every single integer we index into a look-up table of vectors and pull up the word vector for that token**. **Natively transformers processess sets, so we need to positionally encode these vectors, so that we have both the information about the token identity and its place in the sequence from 1 to block size.**

We then add this positional encoding and the token embeddings, as shown in the diagram above (decoder-only image).

x basically contains the set of words and its positions, at the end we feed it to a sequence of blocks (more on that later) and then there's a *layer normalization* (This line applies a **linear transformation** to the output of the last transformer block, denoted by x. This linear transformation is typically a single fully-connected layer that maps the hidden representation of the input sequence to a lower-dimensional space suitable for the output layer) and *we compute the **logits*** (i.e. the unnormalized probabilities of each word in the vocabulary).

```
# forward the GPT model itself
tok_emb = self.transformer.wte(idx) # token embeddings of shape (b, t, n_embd)
pos_emb = self.transformer.wpe(pos) # position embeddings of shape (1, t, n_embd)
x = self.transformer.drop(tok_emb + pos_emb)
for block in self.transformer.h:
    x = block(x)
x = self.transformer.ln_f(x)
logits = self.lm_head(x)
```

```
class Block(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.ln_1 = nn.LayerNorm(config.n_embd)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = nn.LayerNorm(config.n_embd)
        self.mlp = MLP(config)

    def forward(self, x):
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlp(self.ln_2(x))
        return x
```

```
class MLP(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.c_fc    = nn.Linear(config.n_embd, 4 * config.n_embd)
        self.c_proj  = nn.Linear(4 * config.n_embd, config.n_embd)
        self.dropout = nn.Dropout(config.dropout)

    def forward(self, x):
        x = self.c_fc(x)
        x = new_gelu(x)
        x = self.c_proj(x)
        x = self.dropout(x)
        return x
```

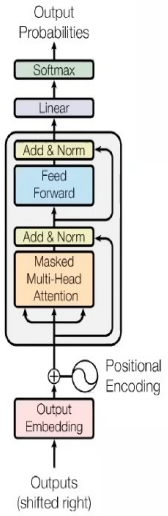
MLP is just a little 2-layer neural network


```python
class CausalSelfAttention(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections for all heads, but in a batch
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd)
        # output projection
        self.c_proj = nn.Linear(config.n_embd, config.n_embd)
        # regularization
        self.attn_dropout = nn.Dropout(config.dropout)
        self.resid_dropout = nn.Dropout(config.dropout)
        # causal mask to ensure that attention is only applied to the left in the input sequence
        self.register_buffer("bias", torch.tril(torch.ones(config.block_size, config.block_size))
                                     .view(1, 1, config.block_size, config.block_size))
        self.n_head = config.n_head
        self.n_embd = config.n_embd

    def forward(self, x):
        B, T, C = x.size() # batch size, sequence length, embedding dimensionality (n_embd)

        # calculate query, key, values for all heads in batch and move head forward to be the batch dim
        q, k ,v  = self.c_attn(x).split(self.n_embd, dim=2)
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)

        # causal self-attention; Self-attend: (B, nh, T, hs) x (B, nh, hs, T) -> (B, nh, T, T)
        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
        att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
        att = F.softmax(att, dim=-1)
        att = self.attn_dropout(att)
        y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
        y = y.transpose(1, 2).contiguous().view(B, T, C) # re-assemble all head outputs side by side

        # output projection
        y = self.resid_dropout(self.c_proj(y))
        return y
```

Stanford

Decoder-only GPT

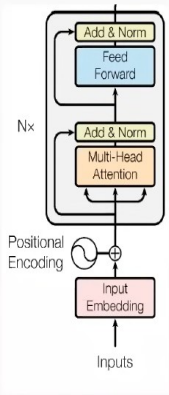
[sat_]



[START] [The_] [cat_]

Encoder-only BERT

[*] [*] [sat_] [*] [the_] [*]



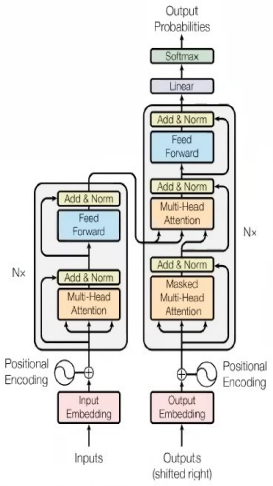
[The_] [cat_] [MASK] [on_] [MASK] [mat_]

Enc- T5

Das ist gut.

A storm in Attala caused 6 victims.

This is not toxic.



Translate EN-DE: This is good.

Summarize: state authorities dispatched…

Is this toxic: You look beautiful today!

Transformer image source: "Attention Is All You Need" paper

Stanford

slide from Lucas Beyer ty