Bertelsmann Technology Scholarship Challenge AI Track 2019/2020 – Course Notes by @Karin

1 Introduction to Neural Networks (Lesson 3)

1.1 Introduction

- Neural Networks mimic how the human brain operates with neurons transmitting information
- Applications: Go, Spam Detection, Stock Price Forecast, Image Recognition, Selfdriving Cars, …

What do Neural Networks do? *They look for the best line/plane/hyperplane to separate data*

	2 dimensional	**3 dimensional**	**n dimensional**
Target	University Acceptance y = label: 0 or 1	University Acceptance y = label: 0 or 1	University Acceptance y = label: 0 or 1
Features	X_1 = test results X_2 = grades	X_1 = test results X_2 = grades X_3 = class rank	X_1 = exam1 X_2 = exam2 X_3 = grades … X_n = essay
Visual			
Boundary Wx + b = 0	Line $w_1x_1 + w_2x_2 + b = 0$	Plane $w_1x_1 + w_2x_2 + w_3x_3 + b = 0$	n-1 dimensional Hyperplane $w_1x_1 + w_2x_2 + \ldots + w_nx_n + b = 0$

Once the boundary is found (i.e. best weights W and bias b is identified), we can predict the score z for a set of x vector data with the boundary equation to determine $\hat{y}$ (check if the the point is above the boundary or below).

$$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$$

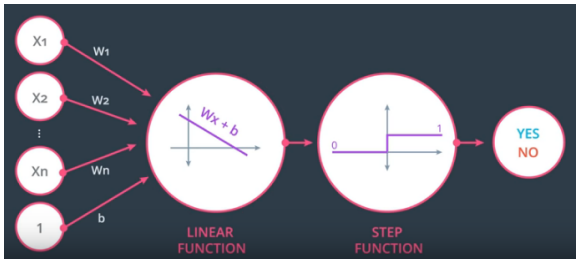
Calculation

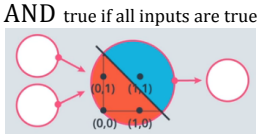
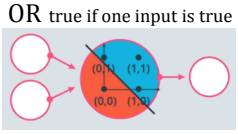
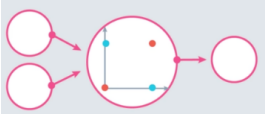
- **weights** W: (1 x n) 1 row, n colums
- **input features** x:(n x 1) n rows, 1 column
- **bias** b: (1x1) scalar

$$z = [w_1 \quad w_2 \quad w_n] \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_n \end{bmatrix} + b = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$$

1.2 Perceptrons

- similar to neurons in the brain
- can be connected to a network so that the output of one perceptron becomes the input of another
- can be used as logical operators (AND → OR: increase both weights or decrease the bias)



AND true if all inputs are true 	OR true if one input is true 
NOT true all inputs are true	XOR* true if exactly one of the inputs is true and the other is false 

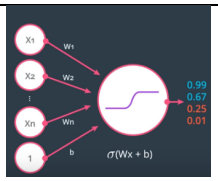
*requires Multilayer Perceptron

Activation Functions *Calculating the Prediction from the Input for Single Layer Networks*

There are many different kind of Activation Functions to compute our prediction $\hat{y}$ from the score. Instead of a *Step Function*, which leads to a **discrete** prediction (yes/no), we should use something that will lead to a **continuous** prediction (probabilities) so that we can better calculate the error:

- *Sigmoid Function* for binary output between 0 and 1 (= special case of the Softmax function)
- *Softmax Function* for outputs with 3 or more classes (each class will get a certain probability)

Calculation

• **score** Z = Wx +b: scalar • **probability** $\hat{y}$: scalar from Sigmoid to Softmax: $y = \frac{1}{1+e^{-x}} = \frac{1}{1+\frac{1}{e^x}} = \frac{1}{\frac{e^x+1}{e^x}} = \frac{e^x}{1+e^x} = \frac{e^x}{e^0+e^x}$	***Binary Problems:*** ***Sigmoid Function*** $\hat{y} = \sigma(Wx+b)$ $\hat{y} = \sigma(Z)$ sigmoid of Z $\hat{y} = \frac{1}{1+e^{-Z}}$  	***Multiclass Problems:*** ***Softmax Function*** $\hat{y}(class\ i) = \frac{e^{Z_i}}{e^{Z_1}+\ldots+e^{Z_n}}$ we could just divide the score of a class by the sum of all scores but that doesn't work for negative scores, so we use exp, which turns the scores into positive numbers first

Cross Entropy *Measuring the Error of a Model*

An idea to measure the model performance compared to actual data: multiply all the predicted probabilities. However, the number would get very small with thousands of probabilities, and a change in one number would have a huge effect on the product. Instead we will take the sum by transforming first to the logarithm. As the natural logarithm of a probability gives a negative number, we take the *negative* natural logarithm to get positive numbers.

- the better the model, the lower the Cross Entropy (negative *ln* of a high probability yields a small number), so our **goal is to minimize the Cross Entropy**
- binary cross entropy is a special case of multiclass cross entropy

Calculation

• **probability** p: scalar • **actual label** y: scalar • **n, m:** number of classes, number of inputs	***Binary Cross Entropy*** $\text{Cross-Entropy} = -\sum_{i=1}^{m} y_i \ln(p_i) + (1 - y_i)\ln(1 - p_i)$	***Multiclass Cross Entropy*** $\text{Cross-Entropy} = -\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij} \ln(p_{ij})$

Error Function *Enables Gradient Descent*

Pre-requisites for an error function for a prediction $\hat{y}$ is that $\hat{y}$ is continuous and differentiable (so this works only for probabilities and not for the yes/no output of a step function).

Calculation

• **actual label** y: scalar • **probability** $\hat{y}$: scalar • **n, m:** number of classes, number of inputs	***Binary Error Function*** $-\frac{1}{m}\sum_{i=1}^{m} y_i \ln(\hat{y}_i) + (1-y_i)\ln(1-\hat{y}_i)$	***Multiclass Error Function*** $-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} y_{ij} \ln(\hat{y}_{ij})$

Gradient Descent *Optimization of the Model for Single Layer Networks*

At each gradient descent step a multiple of the gradient of the error function gets subtracted at every point. This updates the weights and bias.

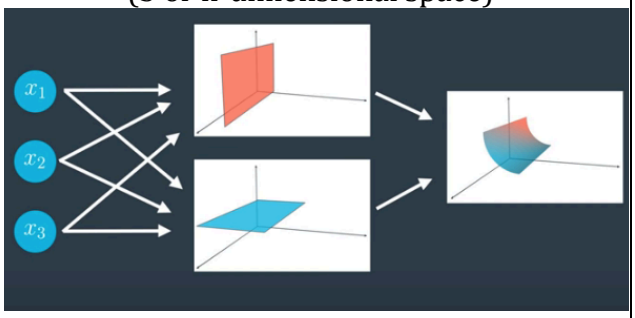
- Gradient is the partial derivative of the error function
- The closer the prediction is to the actual label, the smaller the gradient
- The larger the gradient (i.e. the steeper the error function), the more we'll change our coordinates

Calculation

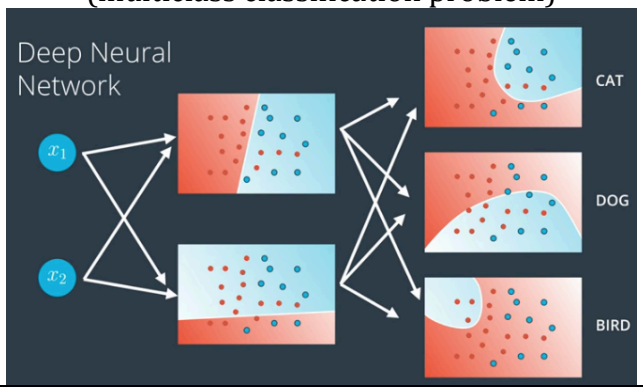
• **actual label** y: scalar • **probability** $\hat{y}$: scalar • **learning rate** α	***Perceptron Algorithm*** 1. Start with random weights: $w_1, \ldots, w_n, b$ 2. For every misclassified point $(x_1,\ldots, x_n)$: a. If prediction == 0: • for i = 1 … n: $w_i = w_i + \alpha$ • $b = b + \alpha$ b. If prediction == 1: • *for i = 1…n:* $w_i = w_i - \alpha$ • $b = b - \alpha$	***Gradient Descent Algorithm*** 1. Start with random weights: $w_1, \ldots, w_n, b$ 2. For every point $(x_1,\ldots, x_n)$: for i = 1 … n: • $w_i = w_i - \alpha(\hat{y}-y)x_i$ • $b = b - \alpha(\hat{y}-y)$ $\frac{\partial E}{\partial w_i} = (\hat{y}-y)x_i$ $\frac{\partial E}{\partial b} = (\hat{y}-y)$ 3. Repeat until error is small

1.3 Multilayer Neural Networks

Input layer having more nodes
(3 or n-dimensional space)

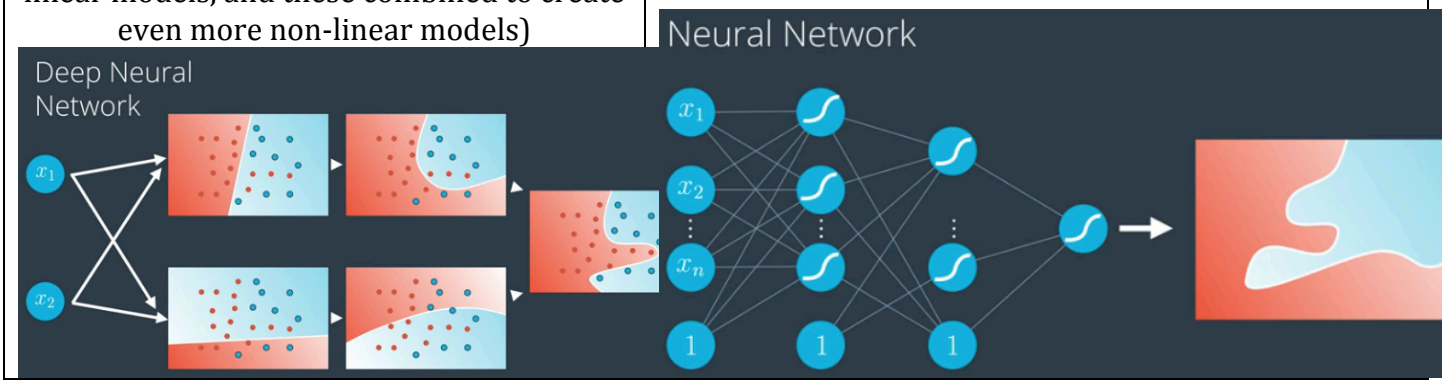


Output layer having more nodes
(multiclass classification problem)



Adding more layers
(linear models combined to create non-linear models, and these combined to create even more non-linear models)

Even more crazy
(highly non-linear boundary in n-dimensional space)



Feedforward Pass *Calculating the Prediction from the Input for a Neural Network*
Feature input X goes through linear transformations (with certain weights and biases) and activation functions and results in a predicted output and a "loss" (prediction error)

Backpropagation *Optimization of the Model for a Neural Network*
We want to find the network parameters where the loss is at the minimum. Backpropagation is the method to minimize the loss for multilayer networks:

1. At each step through the layers we multiply the incoming gradient (derivative) with the gradient of the operation itself – simply applying the **chain rule** (derivative of the derivative).
2. Finally we can subtract the resulting gradient from the weights W to get new weights W'. Those updated weights will result in a lower loss. The learning rate α is set such that the weight update steps are not too large so that the iterative method can actually settle at a minimum.

1.4 Optimization of Neural Network Model Training

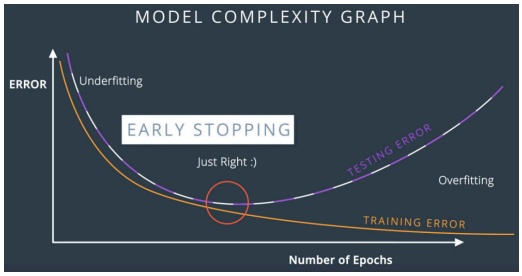
Reasons for a Poor Model:

- Network architecture poorly chosen
- noisy data
- Model running too slow (too small learning rate? – try **Learning Rate Decay** to start with big α and move to smaller α the closer we get to a local minimum
- **Overfitting**: too complex: fits training data well but can't generalize → small training & large test error
- **Underfitting:** too simple/not trained properly → large training error & large test error

Possibile Remedies

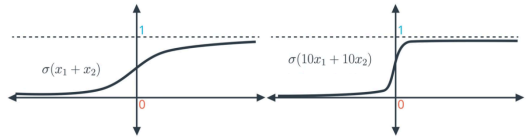
Early Stopping

Stop training when the testing error starts to increase



L1 or L2 Regularization

Penalize large weights by adding a λ term to the error function! This is because large weights cause a steep sigmoid function which makes gradient descent difficult as derivatives are mostly close to 0 and very large at the middle of the curve.



L1 ERROR FUNCTION $= -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)ln(1-\hat{y}_i) + y_i ln(\hat{y}_i) + \boxed{\lambda(|w_1| + ... + |w_n|)}$

L2 ERROR FUNCTION $= -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)ln(1-\hat{y}_i) + y_i ln(\hat{y}_i) + \boxed{\lambda(w_1{}^2 + ... + w_n{}^2)}$

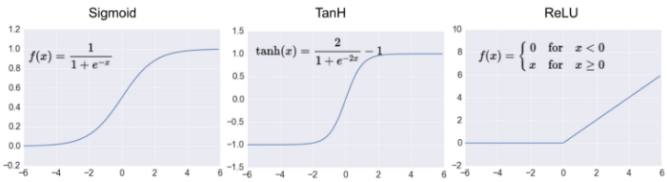
L1: good for feature selection
L2: better for training models

Dropout (= another form of Regularization)

Randomly turn off some nodes during each training epoch for feedforward and backpropagation. Dropout of 0.2 means that each node will be turned off with a probability of 20 %.

Use Other Activation Functions

The derivatives of a Sigmoid function get very small on the right & left which makes training difficult due to tiny changes in the weights.



- **TanH:** similar to Sigmoid but as the range is between -1 and 1 the derivatives are larger
- **Rectified Linear Unit (ReLU):** most common non-linear activation function in hidden layers – for final unit we'd however only use ReLU in case of regression problems! The derivative is 1 if the number is positive. Networks train a lot faster with it.

Random Restart

Prevents getting stuck in Local Minima of the Error Function. There's a higher chance of finding the global minimum by doing gradient descent from a few random places instead of only one.

Momentum

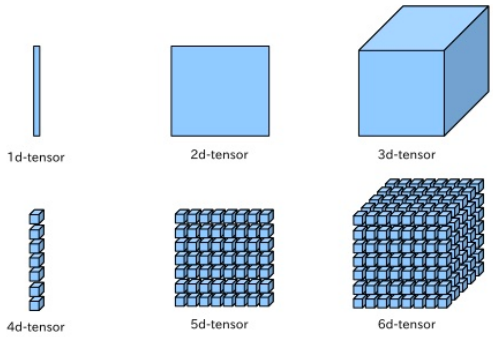
Won't get stuck in local minima during gradient descent (get over the hump with momentum β). How? Weight each step so that the previous step matters more than the one before it.

2 Introduction to PyTorch (Lesson 5)

2.1 Introduction

PyTorch *neural network framework (Python library)*
`import torch`
Developed by Facebook AI team, newer than Keras and Tensorflow & completely open-sourced. Released in Jan 2017. Runs on C++ for the speed but the user interfacing parts are in Python.

Tensors *base data structure in neural network frameworks*
0d-tensor = scalar, 1d-tensor = vector, 2d-tensor = matrix, 3d-tensor = cube. We can imagine a 4d-tensor as a vector of cubes. Tensors can be n-dimensional. Features x, weights w, bias b must have torch tensor format!



Example: A 2d image is stored as a 3d tensor: for every pixel there are three values, one for each color channel (red / green / blue).

- Reminder: for **matrix multiplication**, the number of colums of the first matrix A must match the number of rows of the second matrix B. AB ≠ BA. Reshape tensors if necessary!
- Check if the tensors are the correct shape with the `.shape` method during debugging
- Options for **reshaping** tensors with shape specified:
  - `.reshape(a, b, …)` creates new tensor but might return a clone
  - `.resize_(a, b, …)` in-place operation, changes only the tensor shape but might cut off data if new shape has less or more elements
  - `.view(a, b, …)` creates a new tensor with the specified shape but the old data – returns an error if there is a mismatch in number of elements. Most desireable method to safely change the shape of a tensor! Can use -1 for a dimension to be computed automatically.

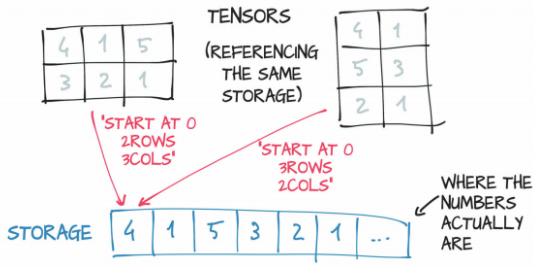


Fig 1: Tensors are views over a Storage Instance
Source: Deep Learning with PyTorch. Stevens, Antiga

Key Formulas in PyTorch Notation

Score ("Logits")	$Wx + b$	`torch.mm(x, w) + b)` *most desirable*
		`torch.matmul(x, w) + b)` *supports broadcasting → error-prone*
		`torch.sum(x * w) + b)`
		`(x * w).sum() + b`

Sigmoid Activation Function	$\frac{1}{1+e^{-z}}$	`1/(1+torch.exp(-z))`
Softmax Activation Function	$\frac{e^{z_i}}{e^{z_1}+\ldots+e^{z_n}}$	`torch.exp(z)/torch.sum(torch.exp(z), dim=1).view(-1, 1)`

Reminder on how Neural Networks work as Universal Function Approximator

During training we pass in many **inputs**, e.g. images, with their actual label (**output**) so that the neural network can build to approximate the function that is converting the images to a probability distribution. How?

1. Initially the weights of the NN are randomly set and after each **forward pass** of data through the network we compare the prediction with the actual label,
2. calculate the prediction error with a **loss function**,
3. change the network weights slightly in the direction that will reduce the error during **backprogation** (by subtracting the gradient multiplied with the learning rate from the weights) and then
4. repeat until the prediction error is small enough.
5. After training, we validate on a separate dataset to see how well the NN generalizes, train/validate again if not happy, and then the NN can be used to make inferences on unseen data.

2.2 Neural Networks in Pytorch

Neural Network Definition in PyTorch

We can use torch.nn, torch.nn.functional or torch.nn.sequential.

Example: network with 784 input units, a hidden layer with 128 units and a ReLU activation, a hidden layer with 64 units and a ReLU activation, and an output layer with softmax.

torch.nn.sequential

```
input_size = 784
hidden_sizes = [128, 64]
output_size = 10

# Build a feed-forward network
model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[1], output_size),
                      nn.Softmax(dim=1))
```

torch.nn.functional

```
class Network(nn.Module):
    def __init__(self):
        super().__init__()
        # Defining the layers, 128, 64, 10 units each
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        # Output layer, 10 units - one for each digit
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        ''' Forward pass through the network'''

        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.softmax(x, dim=1)

        return x

model = Network()
```

Loss Functions in PyTorch

The output of the model has to be adapted depending on which loss function we are using. There are a lot of different loss functions, and in PyTorch there are two possibilities for implementing **negative log-likelihood loss**: CrossEntropyLoss() and NLLLoss(). The latter is preferred. Output is the average loss over the full batch of images that was passed in.

Table 1: example of model & loss function definition for the MNIST problem (handwritten digits recognition) with 784 input units (28x28 pixels)

CrossEntropyLoss()

*Requires the raw **scores** (aka "logits) of the model as input because PyTorch's CrossEntropyLoss internally applies a softmax function first and then NLLLoss.*

```
model = nn.Sequential(   nn.Linear(784, 128)
                         nn.ReLu()
                         nn.Linear(126, 64)
                         nn.ReLu()
                         nn.Linear(64, 10))

images, labels = next(iter(trainloader))
images = images.view(images.shape[0], -1) # flatten to 1D
criterion = nn.CrossEntropyLoss()
logits = model(images)
loss = criterion(logits, labels)
```

NLLLoss()

*Requires **log-softmax output** of the model as input. So, we explicitly apply a softmax activation function in the model definition. Benefit is that we could convert the log-softmax output to actual probabilities with torch.exp(output).*

```
model = nn.Sequential(   nn.Linear(784, 128)
                         nn.ReLu()
                         nn.Linear(126, 64)
                         nn.ReLu()
                         nn.Linear(64, 10)
                         nn.LogSoftmax(dim=1))
images, labels = next(iter(trainloader))
images = images.view(images.shape[0], -1) # flatten to 1D
criterion = nn.NLLLoss()
logps = model(images)
loss = criterion(logps, labels)
ps = torch.exp(logps)
```

Note about nn.LogSoftmax(dim=1): dim=1 means it calculates softmax across the columns instead of the rows, so across each of our examples (rows are examples) and not across each individual feature (columns are features)
Note about the forward pass: it can be written as model(images) or model.forward(images) – it is equivalent!

Backpropagation in PyTorch

Gradient Calculation PyTorch can keep track of all operations done on tensors with the module **autograd**. If autograd is set on a tensor, it can go backwards through each of the operations done and calculate the gradient with respect to the input parameters.

- **Switching autograd on/off:**
  - Turn on autograd during tensor creation: `torch.zeros(1, requires_grad=True)`
  - Turn on autograd on a tensor x at any time: `x.requires_grad_(True)`
  - Turn off autograd for a block of code: `with torch.no_grad():`
    this is useful to speed up & save memory, e.g. in forward passes to make predictions
  - Turn off autograd entirely: `torch.set_grad_enabled(False)`
  - Check if autograd is enabled for a specific tensor x: `x.requires_grad`
- **Calculating the gradient for a tensor z where autograd was switched on:**
  - Run the backward method on z with respect to x (for example the loss): `z.backward()`
  - Calculate the gradient for x: `x.grad()`

Updating the Weights `from torch import optim`

Once the gradient is calculated, we update the weights & biases of the network with an **optimizer** from PyTorch's optim package, e.g. Stochastic Gradient Descent, Adam, Adagrad, NAG, Adadelta,…

- **Define the optimizer** with one of the built in optimizers:
  `optimizer = optim.SGD(model.parameters(), lr = 0.003)` or
  `optimizer = optim.Adam(model.parameters(), lr = 0.003)`, which is like SGD, but it uses momentum to speed up the fitting process. It also adjusts the learning rate for each of the individual parameters of the model
- **Clear the gradients:** PyTorch per default accumulates the gradients, this means they would get summed up in multiple forward/backward passes, so the gradients must be cleared before each training step with `optmizer.zero_grad()`
- **Update the weights** with `optimizer.step()`

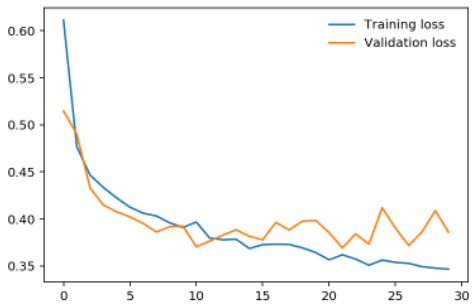
Model Validation *measuring model performance on data that is not part of the training set*

- **Top-5 Error Rate** `.topk()` applied to the probabilities tensor returns a tuple of the *k* highest probabilities and their corresponding class. Usually we are only interested in the highest class:
  top_p, top_class = ps.topk(1, dim=1)
  print(top_class)
- **Accuracy** (% predicted correctly) is checking for how many cases top_class equals the label:
  *equals = top_class == labels.view(*top_class.shape)*
  accuracy = torch.mean(equals.type(torch.FloatTensor))
  print(accuracy)
- **Precision / Recall**

Watchout: If we are using any dropout layers in our model (nn.Dropout(0.2)), we must switch off dropout in the validation pass and for inferences by turning to evaluation mode with `model.eval()`

2.3 Training / Validation Loop

```
model = Classifier()
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.003)
epochs = 30
steps = 0

train_losses, test_losses = [], []
for e in range(epochs):
    running_loss = 0
    # TRAINING LOOP
    for images, labels in trainloader:
        optimizer.zero_grad()
        log_ps = model(images)
        loss = criterion(log_ps, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    # VALIDATION LOOP
    else:
        test_loss = 0
        accuracy = 0
        # Turn off gradients for validation, saves memory and computations
        with torch.no_grad():
            model.eval() # switch to evaluation mode as we are using drop-out in our model
            for images, labels in testloader:
                log_ps = model(images)
                test_loss += criterion(log_ps, labels)

                ps = torch.exp(log_ps)
                top_p, top_class = ps.topk(1, dim=1)
                equals = top_class == labels.view(*top_class.shape)
                accuracy += torch.mean(equals.type(torch.FloatTensor))

        model.train() # switch back to training mode to include drop-out again

        train_losses.append(running_loss/len(trainloader))
        test_losses.append(test_loss/len(testloader))

        print("Epoch: {}/{}.. ".format(e+1, epochs),
              "Training Loss: {:.3f}.. ".format(train_losses[-1]),
              "Test Loss: {:.3f}.. ".format(test_losses[-1]),
              "Test Accuracy: {:.3f}".format(accuracy/len(testloader)))
```



validation & training loss over epochs

Useful Tip:
Using the GPU will speed up computations massively

- Check for GPU availability:
  `torch.cuda.is_available()`
- In any case, make sure the model and all necessary tensors (e.g. images) are moved to either the GPU or CPU with `.to(device)` where `device` is either `"cuda"` or `"cpu"`.

3 Convolutional Neural Networks (Lesson 6)

3.1 Introduction

CNNs are NN that can remember spatial information. They can be trained to identify **features** in an image (shapes/colors) & use them to classify images. Opposite to Multilayer Perceptrons, that work with images flattened to vectors and fully connected / dense layers, CNNs "understand" that pixels in close proximity to each other are more heavily related than pixels that are far apart.
CNNs are several layers of filter operations with *nonlinear activation functions* like ReLU or tanh. During training, **a CNN automatically learns the values of its filters** based on the task you want to perform. For example, in Image Classification a CNN may learn to detect edges from raw pixels in the first layer, then use the edges to detect simple shapes in the second layer, and then use these shapes to deter higher-level features, such as facial shapes in higher layers. Output is a feature level representation of the image (feature vector). The last layer is then a classifier (fully connected layer). The final step can also be used for other tasks, such as generating text – e.g. captions for images.

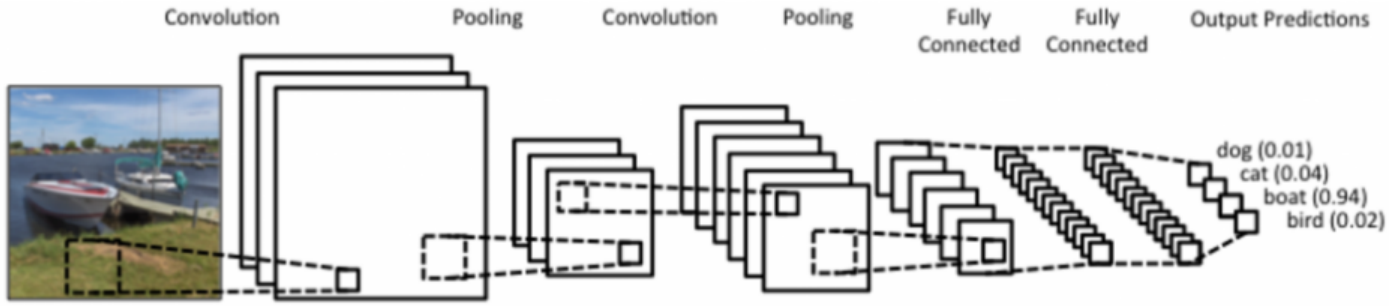


Figure 2: *CNN Architecture Example* **[Souce Link]**

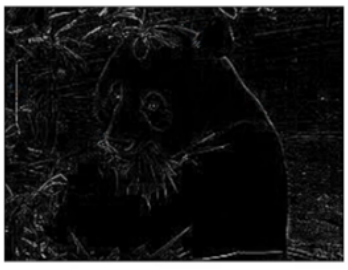
Application Areas
voice user interfaces (e.g. WaveNet Model), **natural language processing:** text classification , language translation, **computer vision:** e.g. classifiying house numbers or traffic signs**, turn 2D images into 3D** by predicting depth

3.2 Filters *Filter out irrelevant information or amplify object boundaries or other distinguishing traits*

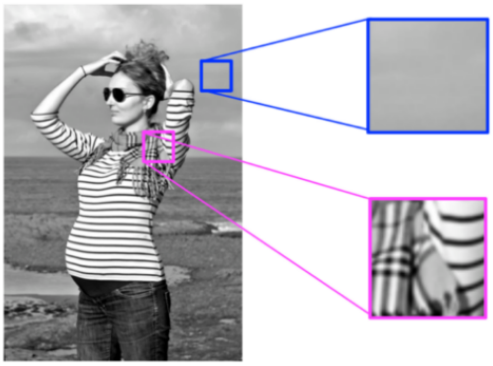
High Pass Filters for Edge Detection

- enhance high frequency parts of an image
- grey out low frequency parts of an image

→ **let an image appear sharper, emphasizes the edges**

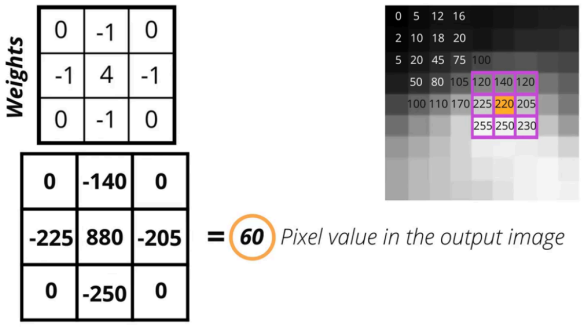


How? Each pixel value gets transformed based on the pixels around it by using filters. Filters have the form of matrices (called "*Convolution Kernels*"). For edge detection, all kernel elements must sum to zero. The kernel is passed over the image pixel by pixel with a certain *stride* size (i.e. "steps" the filter takes).



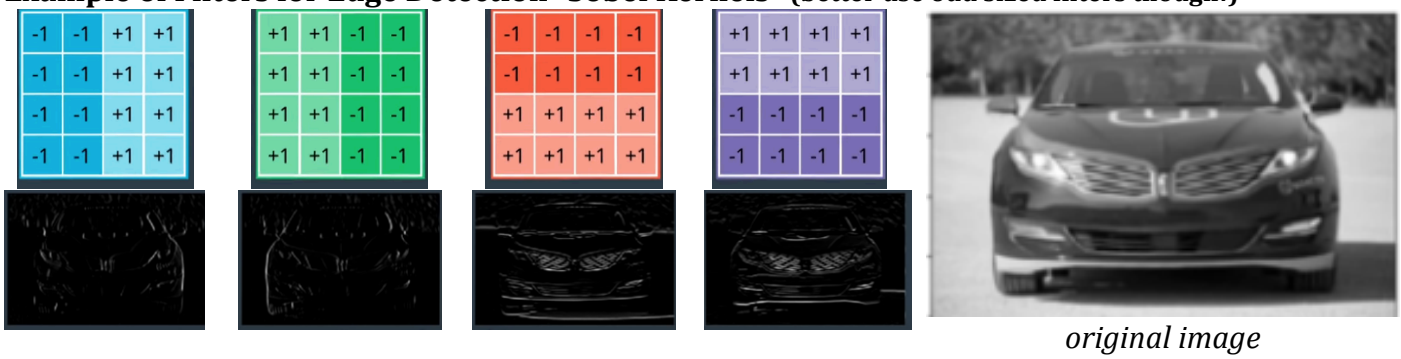
Low Frequency Components
Relatively uniform brightness, slow changes of intensity between pixels
High Frequency Components
Intensity/brightness changes rapidly betw. neighboring pixels (high contrast)

Technical Procedure (Example)
Kernel is put over the pixel so that the pixel is in the center of the kernel. All pixels covered by the kernel are multiplied with their respective weights from the kernel. The sum will be the new value of the center pixel. **In mathematical terms: the image gets "convolved" with the kernel.**
Note: At the edges & corners we have to do padding, cropping or extension. We should also think about kernel size and stride.



- **Padding:** The image is padded with a border of 0's, black pixels.
- **Cropping:** Any pixel in the output image which would require values from beyond the edge is skipped. Can result in the output image being slightly smaller, with the edges having been cropped.
- **Extension:** the border pixels of an image are copied and extended far enough to result in a filtered image of the same size as the original image. Corner pixels are extended in 90° wedges, other edge pixels are extended in lines.

Example of Filters for Edge Detection "Sobel Kernels" (better use odd sized filters though!)

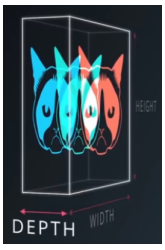


original image

Example of Other Filters – Source: http://setosa.io/ev/image-kernels/

3.3 Layers in CNNs

Convolutional Layer *keep the XY dimensions but increase the depth of an image tensor*
In CNNs, a series of filters (see previous chapter) makes up a convolutional layer. Can be thought of as a stack of filtered images. The neural network will learn the best filter weights as it is trained on a set of image data.



MaxPooling Layer *discard some spatial information*
Pooling (also called downsampling), reduces the (x,y)-dimensions of each feature while maintaining its most important information. The image size is decreased with a filter that commonly has size 2x2 and stride 2 and will make the (x,y)-dimensions half of what they were previously. Makes feature detection more robust to object orientation and scale changes.

Alternatives to MaxPooling

- **Average Pooling** averages pixels in a given window size
- **Capsule Networks** learn spatial relationships between parts (e.g. eyes, nose, mouth) and do not discard spatial information. Capsules are collections of parent & child nodes that build up a complete picture of an object.. Each capsule outputs a vector with **magnitude** *m* (probability that a part exists) & **orientation** θ (state of the part properties).

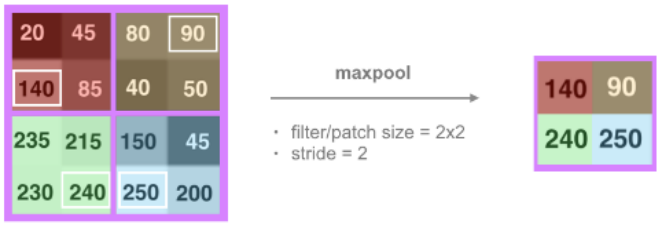
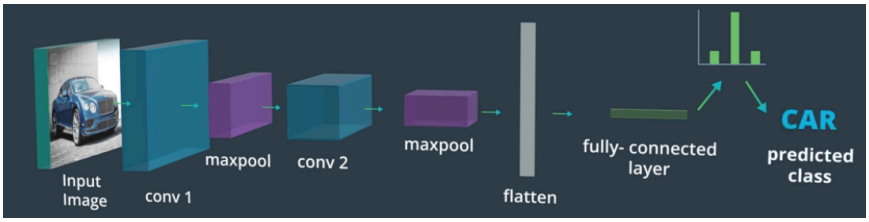


Figure 3: *Max Pooling with 2x2 filter & stride 2*

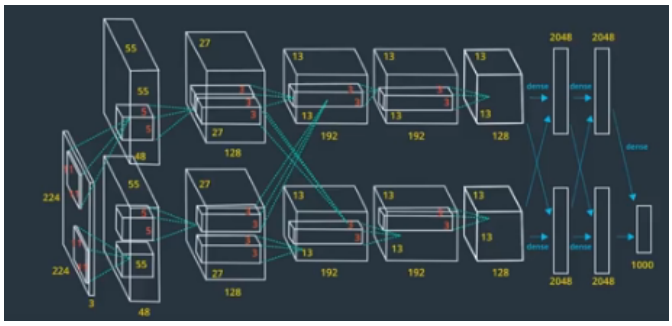
The goal of the alternating sequence of **convolutional & maxpooling layers** is to obtain an array that is very deep but small in the (x,y)-dimensions. This array is the input to a final fully connected layer.



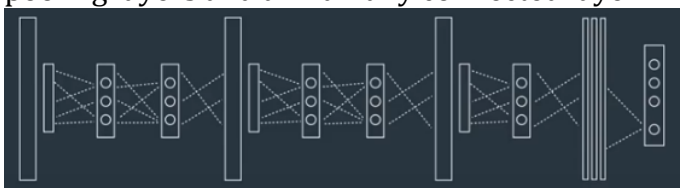
3.4 CNN Architectures

Since 2010 there is an annual competition for CNN architectures for object recognition & classification with ImageNet's database of 1.28 million labelled images from 1,000 categories: *ImageNet Large Scale Visual Recognition Competition.* The best ones are available in PyTorch to use for transfer learning!

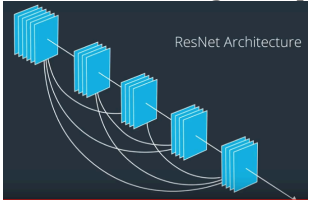
- **AlexNet** - University of Toronto, 2012
  Pioneered ReLu activation function & dropout; 11x11 convolution windows



- **VGGNet** – Visual Geometry Group Oxford Univ., 2014
  Long sequence of 3x3 convolutions with 2x2 pooling layers and a final fully connected layer



- **ResNet** – Microsoft Research, 2015
  Residual learning framework enables exremely deep networks (up to 152 layers) by adding connections that skip layers to avoid the *vanishing gradient* problem for very deep networks during backpropagation.



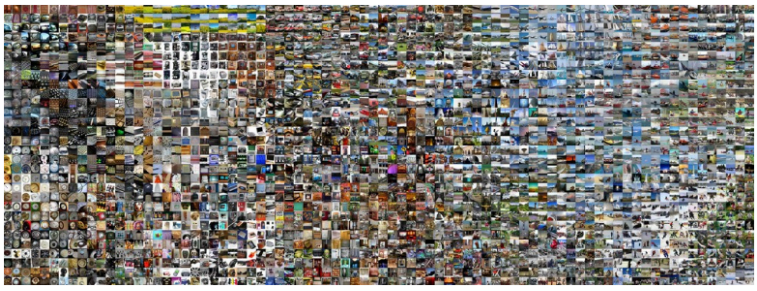
Network	Top-1 error	Top-5 error
AlexNet	43.45	20.91
VGG-11	30.98	11.37
VGG-13	30.07	10.75
VGG-16	28.41	9.62
VGG-19	27.62	9.12
VGG-11 with batch normalization	29.62	10.19
VGG-13 with batch normalization	28.45	9.63
VGG-16 with batch normalization	26.63	8.50
VGG-19 with batch normalization	25.76	8.15
ResNet-18	30.24	10.92
ResNet-34	26.70	8.58
ResNet-50	23.85	7.13
ResNet-101	22.63	6.44
ResNet-152	21.69	5.94
SqueezeNet 1.0	41.90	19.58
SqueezeNet 1.1	41.81	19.38
Densenet-121	25.35	7.83
Densenet-169	24.00	7.00
Densenet-201	22.80	6.43
Densenet-161	22.35	6.20
Inception v3	22.55	6.44

Figure 4: *CNNs pre-trained on ImageNet*

3.5 Visualization of CNNs http://cs231n.github.io/understanding-cnn/

Goal: investigating which part of the image a classification prediction is coming from

- **Layer Activations:** show the activation of the network during forward pass
- **Weights:** visualize the filter weights – typically most interpretable in the first layers
- **Retrieve images that maximally activate a neuron**
- **t-SNE** – Extract the CNN codes (e.g. in AlexNet the 4096-dimensional vector right before the classifier, and crucially, including the ReLU non-linearity) and then plug these into t-SNE and get 2-dimensional vector for each image. The corresponding images can then be visualized in a grid.



- **Occluding parts of an image:** plot the probability of the class of interest (e.g. dog class) as a function of the position of an occluder object. We iterate over regions of the image, set a patch of the image to be all zero, and look at the probability of the class. (see Fig. 4)

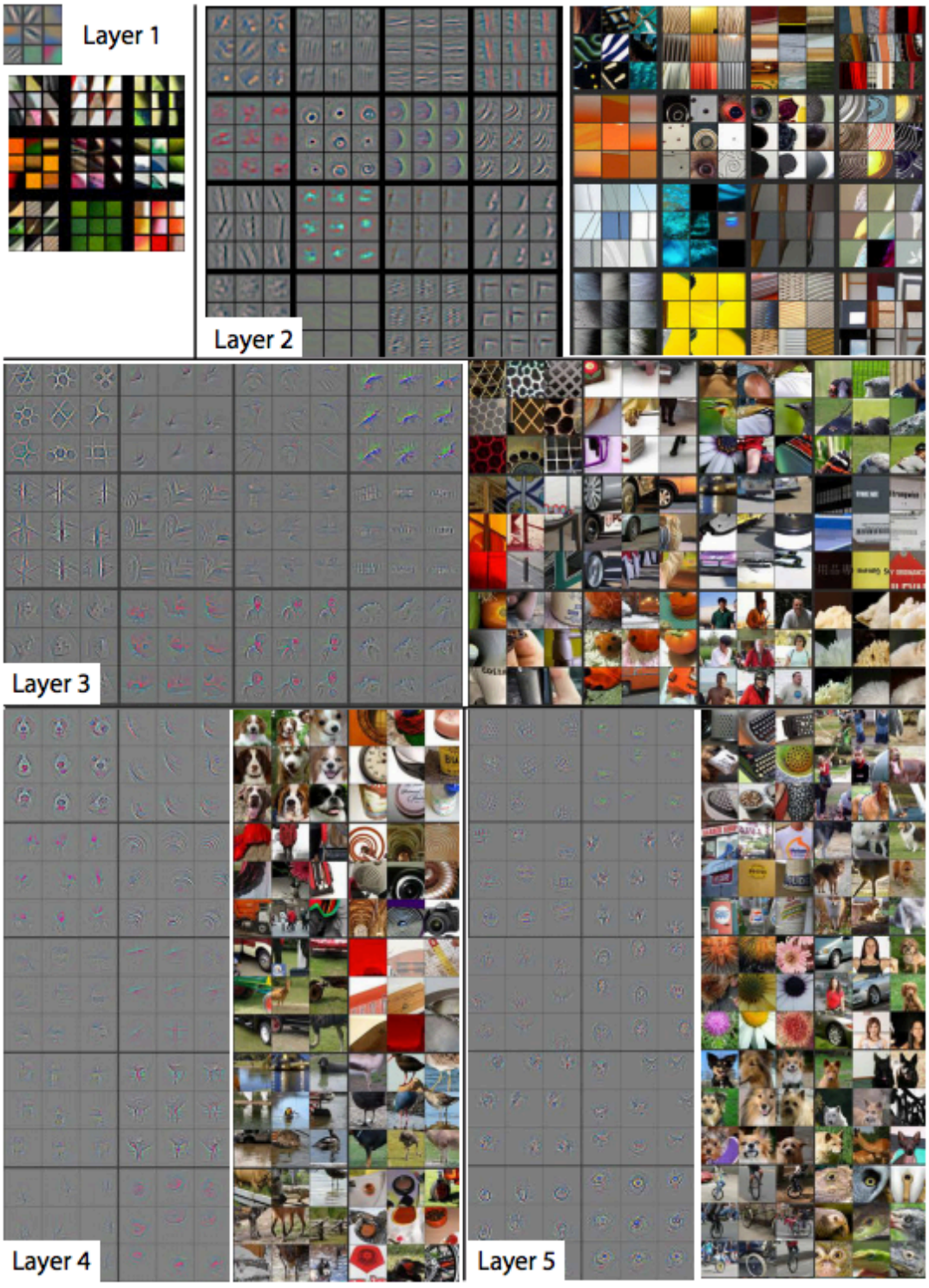


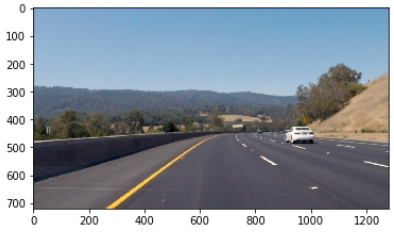
Figure 5: *Visualization of how individual layers of a trained CNN activate based on images* **[Source Link]**

3.6 CNN Implementation in PyTorch

Basic Understanding: Image Visualization / Manipulation with OpenCV `import cv2`

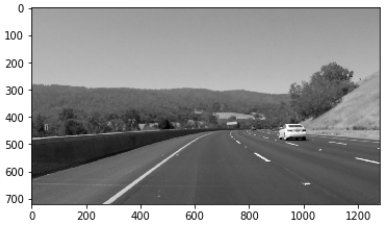
Load & Display Images

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import cv2
import numpy as np
%matplotlib inline

# Read in the image & display it
image = mpimg.imread('data/curved_lane.jpg')
plt.imshow(image)
```

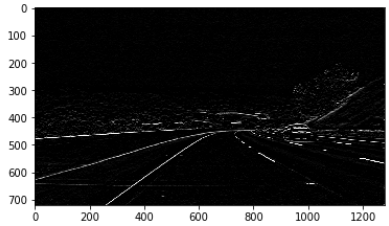


Applying Filters to Grayscale Image (i.e. image with one channel only)

```
# Convert to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
plt.imshow(gray, cmap='gray')
# 3x3 array for edge detection
sobel_y = np.array([[ -1, -2, -1],
                    [ 0, 0, 0],
                    [ 1, 2, 1]])
# Filter the image
# inputs are: grayscale image, bit-depth, kernel
filtered_image = cv2.filter2D(gray, -1, sobel_y)
plt.imshow(filtered_image, cmap='gray')
# print shape and data type of each image
print(gray.shape, type(gray))
print(gray)
print(filtered_image.shape, type(filtered_image))
print(filtered_image)
```



```
(720, 1280) <class 'numpy.ndarray'>
[[ 53  52  29 ...  33  50  53]
 [ 55  33  98 ...  88  34  54]
 [ 43  73 154 ... 141  69  46]
 ...
 [ 47  55 100 ...  67  87 130]
 [ 52  36  68 ...  81 113 142]
 [ 51  55  37 ... 113 138 135]]
```



```
(720, 1280) <class 'numpy.ndarray'>
[[  0   0   0 ...   0   0   0]
 [ 22 157 255 ... 255 139  24]
 [118 215 230 ... 209 200 112]
 ...
 [  0   0   0 ...  49 104 118]
 [  8   0   0 ... 157 153 112]
 [  0   0   0 ...   0   0   0]]
```

CNN Definition in PyTorch

Note: the more convolutional layers we include, meaning the deeper the net, the more complex patterns in color and shape a model can detect. Watchout: problem of vanishing gradient!

Example: a CNN for images of size 32x32, with 3 convolutional layers + ReLU activation (to turn all negative pixels to 0 – i.e. black), max pooling, as well as two linear layers + dropout in between to avoid overfitting.

Note about x.view(-1, 1024): As linear layers always expect the input to be 2d arrays, we flatten the output of the convolutions prior to passing it to the first fully connected layer.

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=1024, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=10, bias=True)
  (dropout): Dropout(p=0.25)
)
```

Computation of the **output size of a given convolutional layer** (i.e. how many neurons define the output) can be done with this formula: [(W−F+2P)/S]+1, where

- W is the input volume size,
- F is the kernel/filter size,
- S is the stride with which they are applied, and
- P is the amount of zero padding used on the border.

For example for a 7x7 input and a 3x3 filter with stride 1 and pad 0 we would get a 5x5 output. [(7 – 3 + 2*0) / 1] + 1 = 5. With stride 2 we would get a 3x3 output: [(7 – 3 + 2*0) / 2] + 1 = 3.

```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # convolutional layer (sees 32x32x3 image tensor)
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        # convolutional layer (sees 16x16x16 tensor)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        # convolutional layer (sees 8x8x32 tensor)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # linear layer (64 * 4 * 4 -> 500)
        self.fc1 = nn.Linear(64 * 4 * 4, 500)
        # linear layer (500 -> 10)
        self.fc2 = nn.Linear(500, 10)
        # dropout layer (p=0.25)
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        # add sequence of convolutional and max pooling layers
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        # flatten image input
        x = x.view(-1, 64 * 4 * 4)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add 2nd hidden layer, with relu activation function
        x = self.fc2(x)
        return x

# create a complete CNN
model = Net()
```

Convolutional Layers in PyTorch

To create a convolutional layer in PyTorch, you must first import the necessary module:

```
import torch.nn as nn
```

Then, there is a two part process to defining a convolutional layer and defining the feedforward behavior of a model (how an input moves through the layers of a network. First you must define a Model class and fill in two functions.

init

You can define a convolutional layer in the `__init__` function of by using the following format:

```
self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)
```

forward

Then, you refer to that layer in the forward function! Here, I am passing in an input image `x` and applying a ReLu function to the output of this layer.

```
x = F.relu(self.conv1(x))
```

Arguments

You must pass the following arguments:

- `in_channels` - The number of inputs (in depth), 3 for an RGB image, for example.
- `out_channels` - The number of output channels, i.e. the number of filtered "images" a convolutional layer is made of or the number of unique, convolutional kernels that will be applied to an input.
- `kernel_size` - Number specifying both the height and width of the (square) convolutional kernel.

There are some additional, optional arguments that you might like to tune:

- `stride` - The stride of the convolution. If you don't specify anything, `stride` is set to `1`.
- `padding` - The border of 0's around an input array. If you don't specify anything, `padding` is set to `0`.

NOTE: It is possible to represent both `kernel_size` and `stride` as either a number or a tuple.

There are many other tunable arguments that you can set to change the behavior of your convolutional layers. To read more about these, we recommend perusing the official documentation.

Pooling Layers

Pooling layers take in a kernel_size and a stride. Typically the same value, the is the down-sampling factor. For example, the following code will down-sample an input's x-y dimensions, by a factor of 2:

```
self.pool = nn.MaxPool2d(2,2)
```

forward

Here, we see that poling layer being applied in the forward function.

```
x = F.relu(self.conv1(x))
x = self.pool(x)
```

3.7 Image Classification with Transfer Learning – Step by Step

Create Folders with Training & Validation Images

Images must be in a **folder structure** with one folder per class (example to the right). The folder name per class becomes the label of each image.

```
root/dog/xxx.png
root/dog/xxy.png
root/dog/xxz.png

root/cat/123.png
root/cat/nsdf3.png
root/cat/asd932_.png
```

Transform Images

1. **Resize** all images to the same size (adjust pixel size / crop the center)
2. **Data Augmentation** as needed (on training images only!) to introduce some randomness
3. Convert images to PyTorch **Tensor**
4. **Normalize** the color channels (subract mean & divide by standard deviation)

All transforms can be combined into a pipeline with `transforms.Compose()`

Load Images

The DataLoader takes a dataset and returns batches of images and the corresponding labels (the label is taken from the folder structure!). It is a **generator**. To get data out of it, you need to loop through it or convert it to an **iterator** and call next().

Load a Pre-Trained Model for Transfer Learning

We **select a pre-trained network** (e.g. VGGNet or DenseNet), **define a classifier, set the criterion and optimizer**. Then we load the model and **freeze the weights** for the whole network except that of the final fully connected layer. This last fully connected layer is replaced with our own classifier with random weights and only this layer is trained in the next step. Note about the classifier: the number of **input features** of the classifier depend on the pretrained model (e.g. 25088 for VGG and 1024 for DenseNet), the number of **output features** depends on the number of classes we're trying to predict.

Note: Such networks were trained on images on ImageNet – a collection of 1 million labelled images from 1,000 categories. Most of the pretrained models require the input to be 224x224 images.

Train the Model (Classifier only)

Training is an iterative process and is done over a series of **Epochs**. Each epoch consists of a training part and evaluation part. We stop the iteration when we're happy with the model's accuracy.

Training Mode: The algorithm computes the output predictions by passing inputs (images) to the model (**forward pass**). Then the batch **loss** is calculated by comparing vs. the actual labels. The gradient of the loss is computed with respect to model parameters via a **backward pass**. Finally, the parameters are updated in an **optimization** step.

Evaluation Mode: The model is asked to make a prediction on validation images with a **forward pass**. Then the batch **loss** is calculated by comparing vs. the actual labels. The optimizer is turned off completely. **Performance measures** are typically *accuracy* (= percentage of classes the network predicted correctly), *precision*, *recall*, *top-5 error rate*.

Could consider to "Retrain" to get an even higher accuracy after training the classifier. We could unfreeze the whole model and train the whole model and classifier again with a small Learning Rate. The very small learning rate of 0.0001 or smaller will tune the model to work better with a specific dataset.

Save Model / Load Model

After having trained a model, we can save it for later fine-tuning or for making predictions. As the tensors for weights and biases are stored in model.state_dict(), we can save this and name the file:

- Saving: `torch.save(model.state_dict(), "checkpoint.pth")`
- Loading: `torch.load("checkpoint.pth")`

The model however must have the same architecture (same number of input/hidden/output layers) as the network that the checkpoint was created with. So, ideally we also save the model architecture in a dictionary with all the necessary information to re-build the model.

Make Predictions

An image is pre-processed so it fullfills the model requirements (correct size & transformed to a tensor) and then passed into the network (forward pass) to predict the most likely class it belongs to.

4 Style Transfer (Lesson 7)

4.1 Introduction

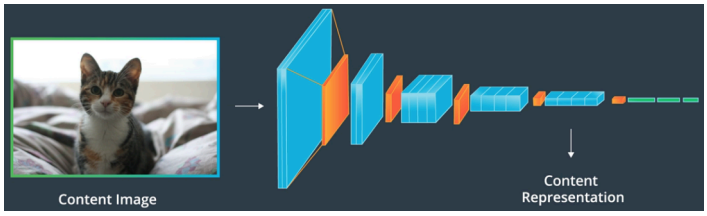
The feature level representations output of CNNs are not only useful for classification but for image construction as well. They are the basis for applications like Style Transfer, & Deep Dream which compose images based on CNN layer activations and extracted features. Style Transfer means applying the style of one image to another image.

4.2 Extracting Content and Style from Images for Style Transfer

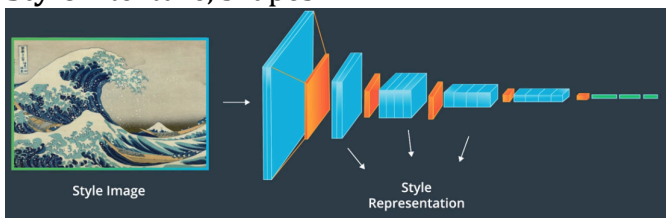
Target

An image gets passed through a trained CNN (e.g. VGG19) in a feedforward pass. Content and Style get extracted from different layers in the network: content from the deeper layers, style (shapes, colors) from the first layers. So, we are using the network as a **feature extractor**.

Content



Style – texture, shapes




```
conv1_1  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (1): ReLU(inplace=True)
         (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (3): ReLU(inplace=True)
         (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
conv2_1  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (6): ReLU(inplace=True)
         (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (8): ReLU(inplace=True)
         (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
conv3_1  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (11): ReLU(inplace=True)
         (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (13): ReLU(inplace=True)
         (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (15): ReLU(inplace=True)
         (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (17): ReLU(inplace=True)
         (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
conv4_1  (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (20): ReLU(inplace=True)
conv4_2  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (22): ReLU(inplace=True)
         (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (24): ReLU(inplace=True)
         (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (26): ReLU(inplace=True)
         (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
conv5_1  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (29): ReLU(inplace=True)
         (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (31): ReLU(inplace=True)
         (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (33): ReLU(inplace=True)
         (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (35): ReLU(inplace=True)
         (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

Figure 6: VGG19 contains of 5 stacks of convolutional layers and 3 fully connected layers. According to Gatys paper the first convolutional layer in each stack is used for style representation. The second layer in the fourth stack conv4_2 is used for content representation.

4.3 Loss Calculation *difference in content/style between target and source images*

Gram Matrix

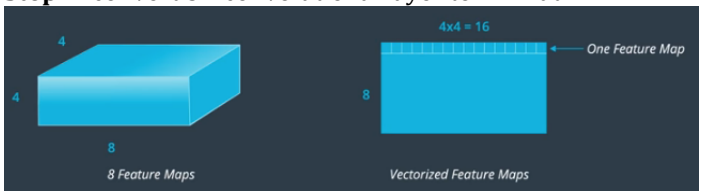
The style representation in images relies on looking at correlations between the features in individual layers of the VGG Network: how similar are the features in a single layer? By including the correlations between layers of different sizes we get a multiscale style represenation of an image, which includes large and small style features.

The correlations in each layer are given by the Gram Matrix. The Gram Matrix contains non-localized information about each layer (prominent colors & textures → the style). The values indicate the similarities between the layers.

Creating a Gram Matrix:

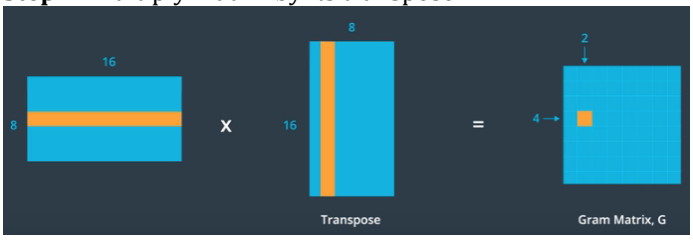
1. Flatten the xy dimensions of the feature maps of a convolutional layer (i.e. convert a 3D convolutional layer into a 2D matrix of values)
2. Multiply this matrix by its transpose (which means multiplying the features in each map (treats each value in each feature map into an individual sample unrelated in space to other values)

Step 1: convert 3D convolutional layer to 2D matrix



d x h x w = 8 x 4 x 4 becomes a matrix of shape: h x w = 8 x 16

Step 2: multiply matrix by its transpose



Gram matrix has shape 8 x 8 to show the correlations between the 8 feature maps of the convolutional layer. Its shape only depends on number of feature maps but not image size! The highlighted cell holds the value indicating the similarities between the 4th and 2nd feature map

Loss Minimization

Content Loss

Mean square distance between target and content features

$$\mathcal{L}_{content} = \frac{1}{2}\sum(T_c - C_c)^2$$

Style Loss

Mean square distances between style and target image gram matrices (all pairs from each convolutional layer we decide to consider for style)
a: number of values in each layer
w: style weights

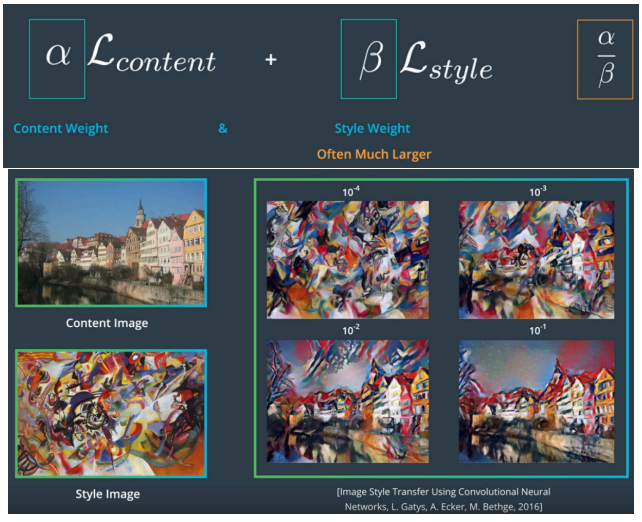
$$\mathcal{L}_{style} = a\sum_i w_i(T_{s,i} - S_{s,i})^2$$

Total Loss

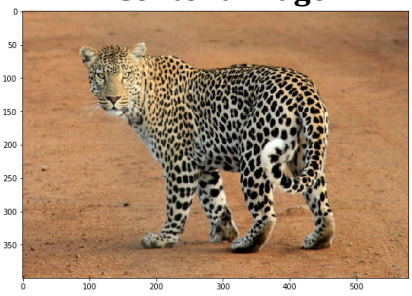
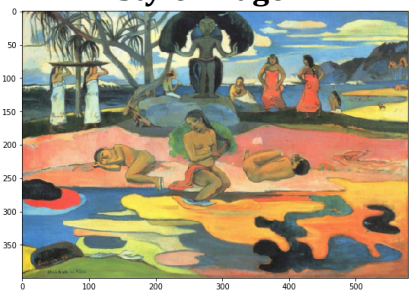
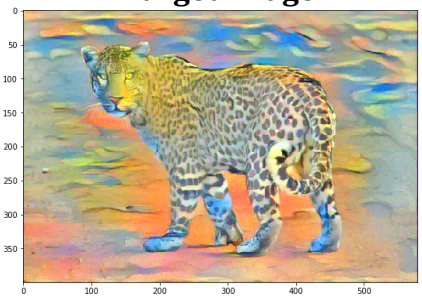
Both losses are added to get the Total Loss. Use backpropagation and optimization to iteratively reduce this loss by changing the target image to match the desired content and style.

As both losses are calculated differently, weight constants α and β are added. The weight for the style loss β is often much larger. The ratio of α/β affects how stylized the image appears.

The smaller the α/β ratio, the more stylistic effect (see figure to the right for ratios from 10^{-1} to 10^{-4}):



4.4 Examples

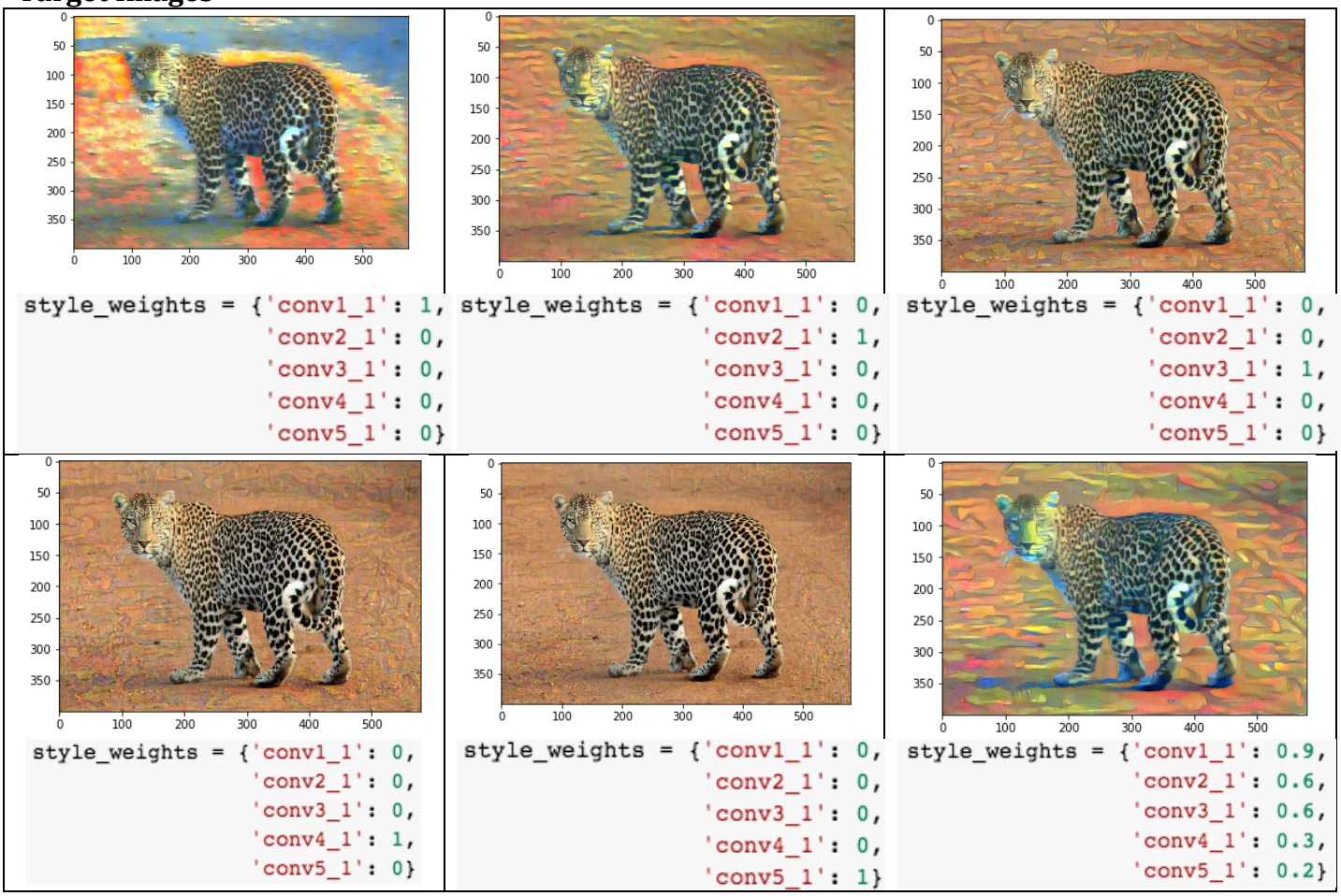
Content Image	Style Image	Target Image*
		

*final settings: α/β ratio 5:1; starting with blank canvas for the target image; style weights: 0.9/0.6/0.6/0.3/0.3; 5000 iterations; conv3_2 as content representation

How do different style weights impact the target image?

Style transfer was run with 3000 iterations of updating the target image (starting with a copy of the content image – but could start with a blank image). As a ratio between content weight and style weight I chose 1:1,000,000. The smaller this ratio, the more stylized the final image is.
By weighting earlier layers (conv1_1 and conv2_1) more, we get larger style artifacts in the target image. If we weight later layers, we get more emphasis on smaller features. This is because each layer is a different size and together they create a multi-scale style representation!

Target Images



```
style_weights = {'conv1_1': 1,
                 'conv2_1': 0,
                 'conv3_1': 0,
                 'conv4_1': 0,
                 'conv5_1': 0}
```

```
style_weights = {'conv1_1': 0,
                 'conv2_1': 1,
                 'conv3_1': 0,
                 'conv4_1': 0,
                 'conv5_1': 0}
```

```
style_weights = {'conv1_1': 0,
                 'conv2_1': 0,
                 'conv3_1': 1,
                 'conv4_1': 0,
                 'conv5_1': 0}
```

```
style_weights = {'conv1_1': 0,
                 'conv2_1': 0,
                 'conv3_1': 0,
                 'conv4_1': 1,
                 'conv5_1': 0}
```

```
style_weights = {'conv1_1': 0,
                 'conv2_1': 0,
                 'conv3_1': 0,
                 'conv4_1': 0,
                 'conv5_1': 1}
```

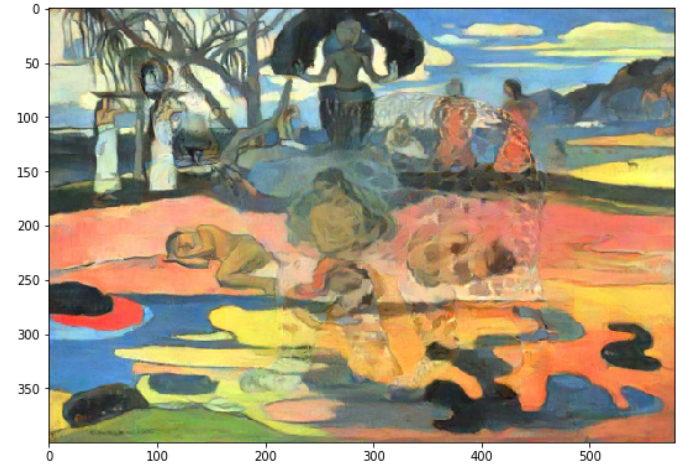
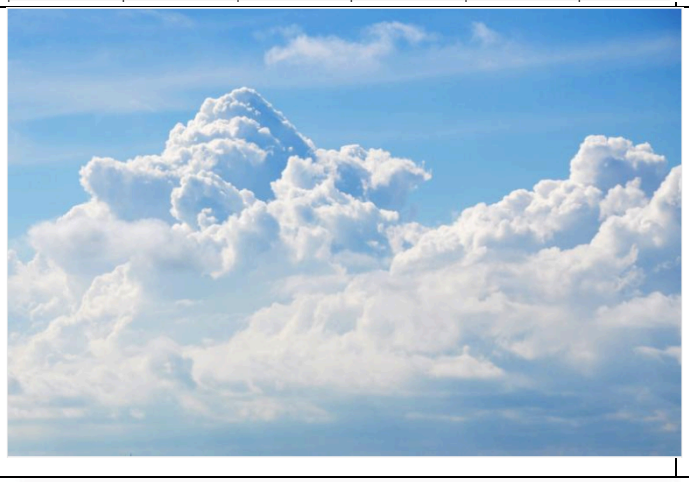
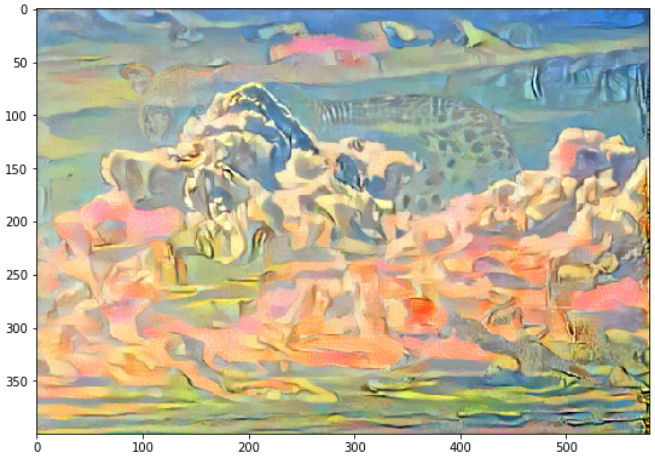
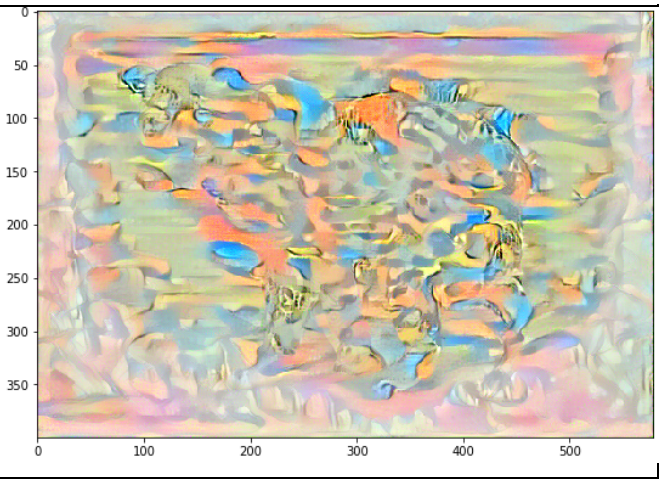
```
style_weights = {'conv1_1': 0.9,
                 'conv2_1': 0.6,
                 'conv3_1': 0.6,
                 'conv4_1': 0.3,
                 'conv5_1': 0.2}
```

What is the impact of the starting point for the target image, # iterations, α/β ratio?

Style transfer was run with 3000 iterations of updating the target image and an α/β ratio of 1:1. Style weights of the individual layers was 1 for all.

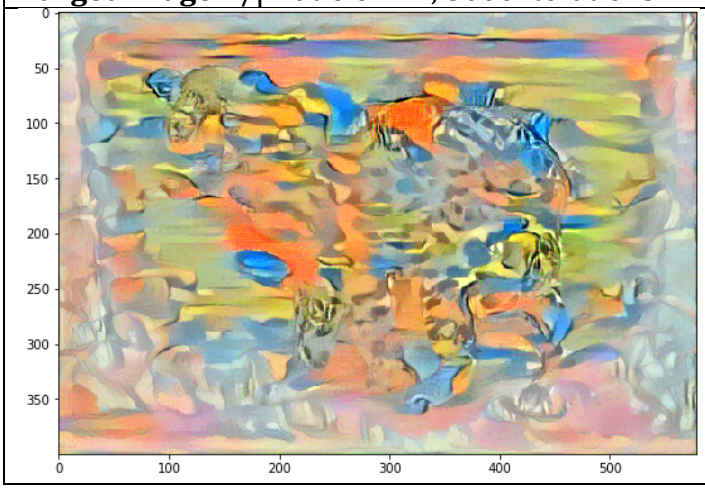
1) a copy of the style image as target image starting point
2) an image of a cloud as target image starting point
3) a blank canvas as target image starting point

```
style_weights = {'conv1_1': 1,
                 'conv2_1': 1,
                 'conv3_1': 1,
                 'conv4_1': 1,
                 'conv5_1': 1}

content_weight = 1  # alpha
style_weight = 1  # beta (1e6)
```

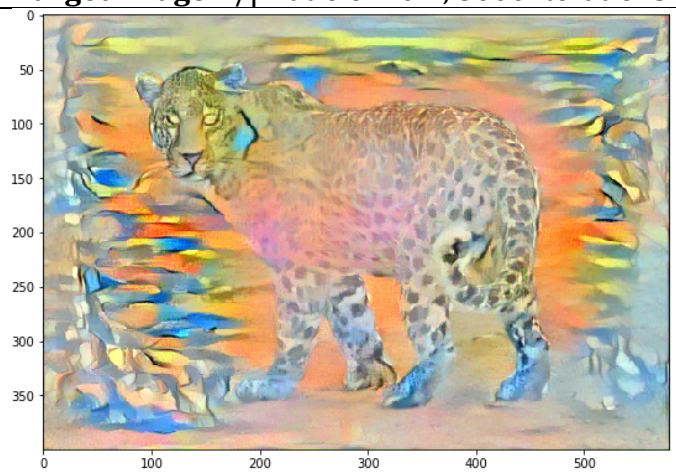
	Starting Point for Target Image	Target Image
1)		
2)		
3)		

The more detailed the starting point of the target image is, the more difficult it is for the content image to "shine through". A blank canvas (3) works well but in oder to display the content better it seems to require a higher number of terations or a higher α/β ratio (weight the content loss higher).

Target Image α/β ratio of 1:1, 5000 iterations



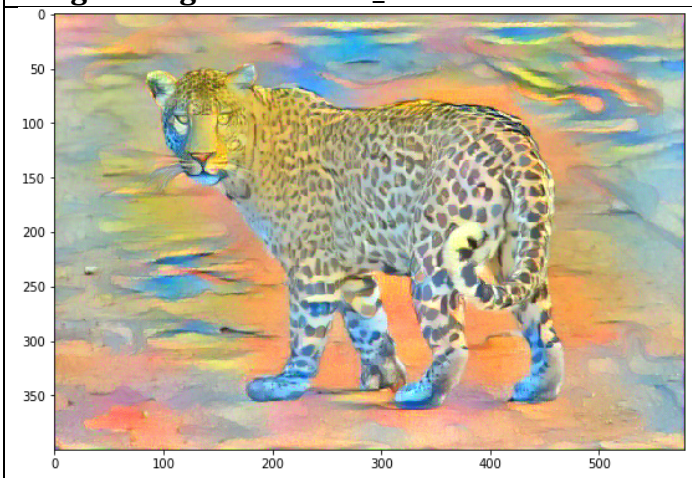
Target Image α/β ratio of 10:1, 5000 iterations



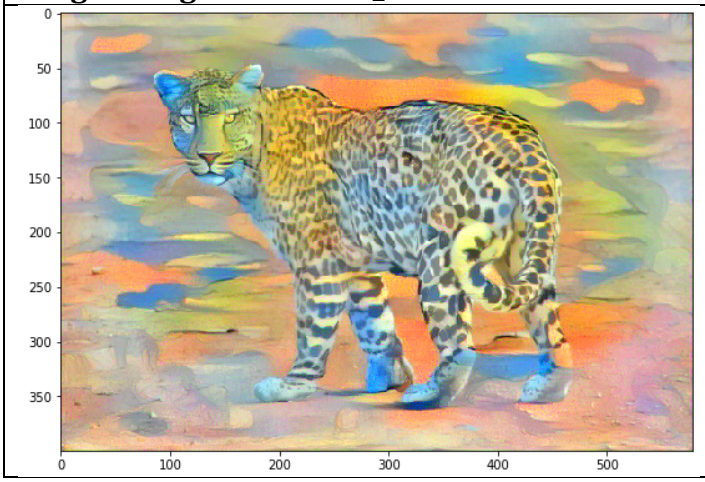
Building on above version with α/β ratio of 10:1 and 5000 iterations, as a final experiment, we will use conv3_2 / conv2_2 / conv1_2 as a content representation instead of conv4_2.

<u>Conclusion: conv3_2 seems to work best</u> and better than conv4_2. Conv2_2 makes the face more harsh. Conv1_2 is not sufficient as it's missing details such as the ears.

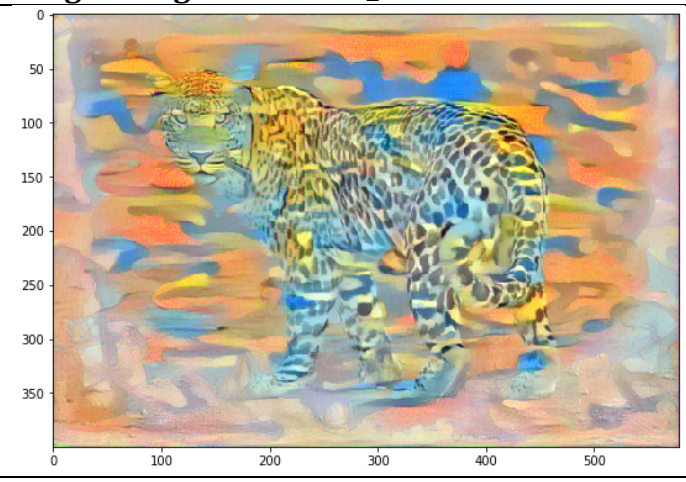
Target Image with conv3_2



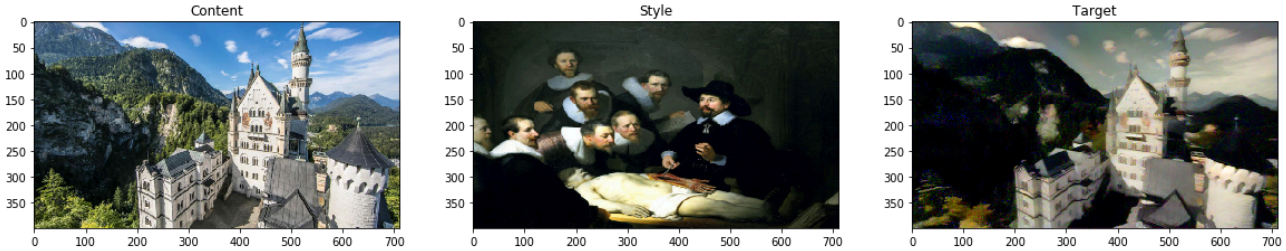
Target Image with conv2_2



Target Image with conv1_2



More Examples

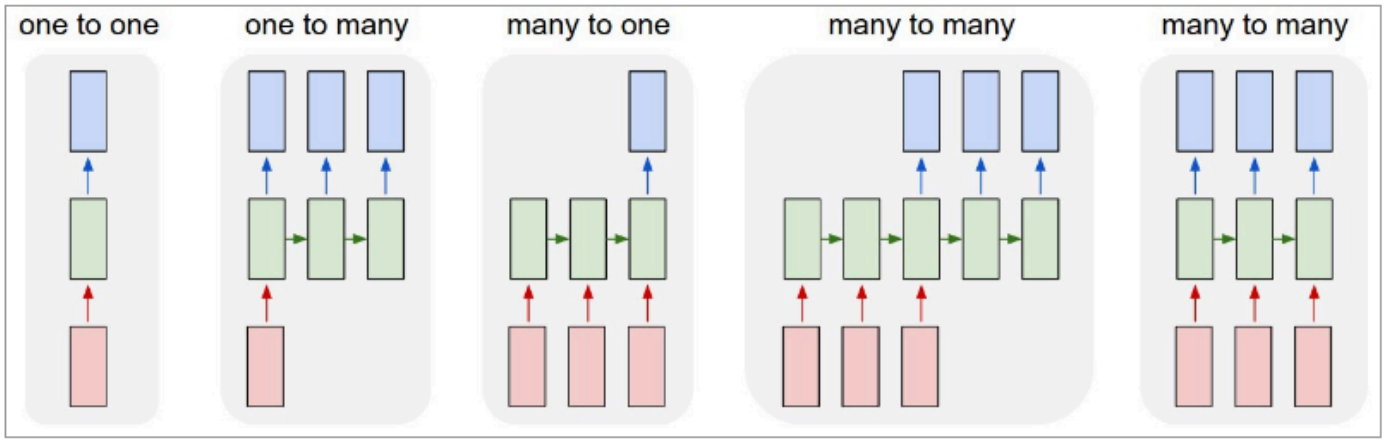


5 Recurrent Neural Networks (Lesson 8)

5. 1 Introduction

- **RNN**s (Recurrent Neural Networks) are designed specifically to learn from sequences of data by passing the hidden state from one step in the sequence to the next step in the sequence, combined with the input. As they don't store long term memory well, we should rather use **LSTMs** or **GRUs**:
- **LSTM**s (Long Short-Term Memory) are an improvement of RNNs as they can switch between remembering recent things, and things from long time ago.
- **GRU**s (Gated Recurrent Unit)

RNNs offer a lot of flexibility as we can operate with sequences at the input/output or both:



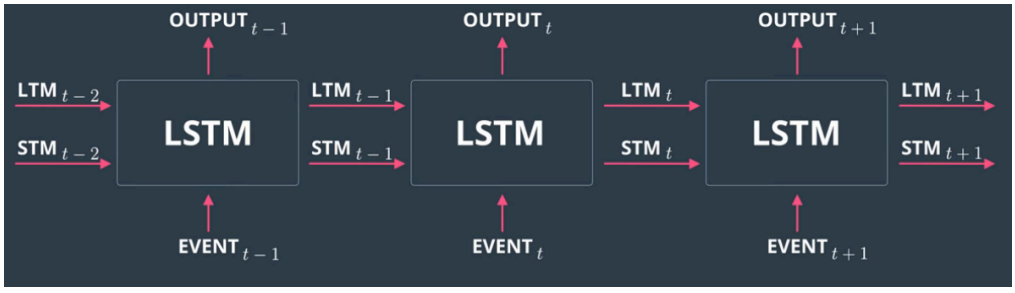
Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state (more on this soon). From left to right: **(1)** Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification). **(2)** Sequence output (e.g. image captioning takes an image and outputs a sentence of words). **(3)** Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). **(4)** Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). **(5)** Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

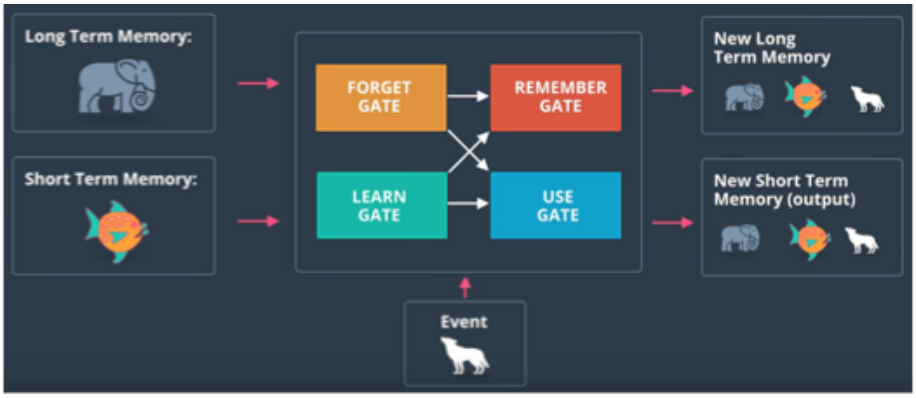
source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

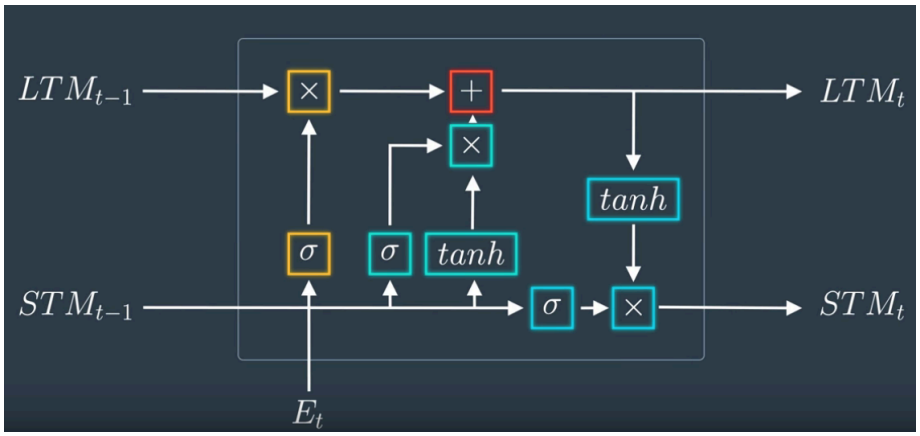
5.2 LSTMs

Topline Understanding

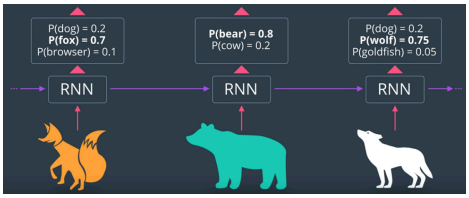
- at every time step, long-term and short-term memory is used as input and it also outputs both
- uses sigmoid and tanh activation functions







Example:
A TV show about nature is showing frames with different animals. The current image shows a wolf that get's recognized by the neural network as a dog (p=0.8), wolf (p=0.15), fish (p=0.05). However, since the previous image was a bear, and before that there was a fox we want that information to influence the prediction. So, with an LSTM the output of the previous prediction is used as part of the input for the next one, and the prediction will be improved:



Forget Gate: takes the long term memory as an input and outputs a new long term memory (only the information it considers useful – forgetting everything else)
Learn Gate: takes short term memory and the event itself as an input (i.e. the information recently learnt) and outputs a new short term memory without any unnecessary information
Remember Gate: takes as input the long term memory (output of forget gate) and the new information (output of the learn gate) joins both and stores it as the new long term memory
Use Gate: decides which information to use from what we previously knew plus from what we just learnt (outputs of forget gate & learn gate) to make a prediction – the output becomes the both the prediction and the new short term memory

5.3 Other Architectures

GRUs

- combines the *forget* and the *learn* gate to an *update* gate
- only returns one working memory instead of short-term and long-term memory

LSTMs with Peephole Connections

- Long term memory has more access to decisions made inside the LSTM by connecting into all of the forget type nodes of the forget gate, learn gate, and use gate

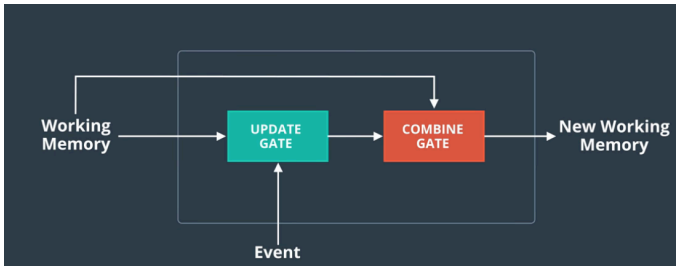


Figure 7: Gated Recurrent Unit (GRU)

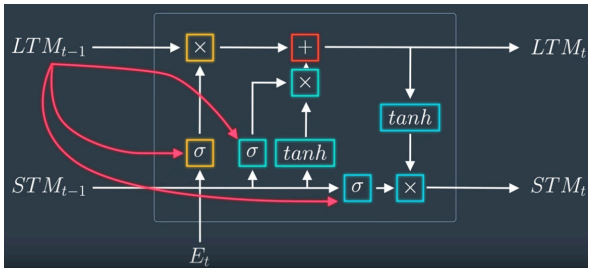


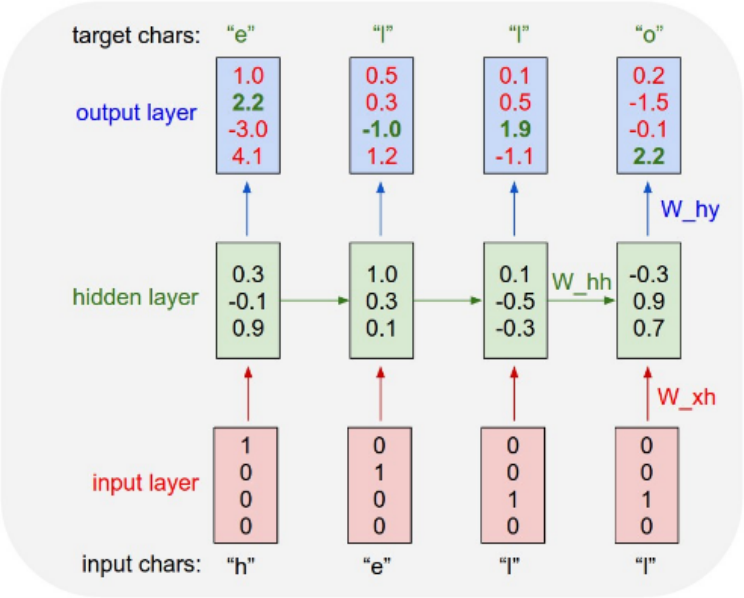
Figure 8: LSTM with Peephole Connection

5.4 Application of LSTMs for Character-Level Language Models

Objective: pass a long text into an RNN one character at a time and ask it to model the probability distribution of the next character in the sequence given a sequence of previous characters. This will allow us to generate new text one character at a time

1. one-hot-encode an input character
2. feed one-hot vector into hidden layer
3. hidden layer has two outputs:
    a. RNN output → goes to final fully connected output layer, which produces a series of character-class scores that is as long as the input vector; it will be passed into a softmax function that will compute a probability distribution for the next most likely character
    b. Hidden state → will be fed into the hidden layer at the next time step in the sequence

The LSTM accepts an input vector `x` and gives you an output vector `y`. The output vector's contents are influenced not only by the input you just fed in, but also on the entire history of inputs you've fed in in the past.
So, if a cell has just generated the character "a", it likely will *not* generate another "a" right after that.

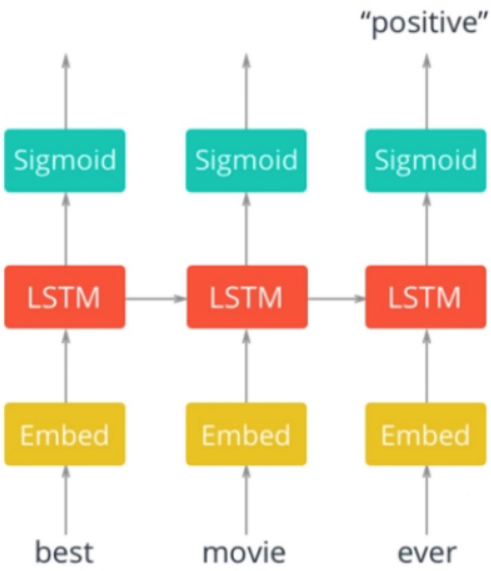


An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons). This diagram shows the activations in the forward pass when the RNN is fed the characters "hell" as input. The output layer contains confidences the RNN assigns for the next character (vocabulary is "h,e,l,o"); We want the green numbers to be high and red numbers to be low.
http://karpathy.github.io/2015/05/21/rnn-effectiveness/

6 RNN's for Sentiment Prediction (Lesson 9)

6.1 Introduction

Objective: develop a neural network that can classify the sentiment of a movie review into "positive" or "negative" by training it on a dataset of movie reviews from IMDB that have been labeled as positive or negative.
Method: As this is text data (i.e. words in a sequence), we can use an RNN to build a model that doesn't only consider the individual words, but the order they appear in.



1. **Load in text data**
2. **Data Pre-Processing** *encode characters as integers*
3. **Data Padding & Truncation** *standardize the sequence length of each review*
4. **Create Tensor Dataset & DataLoader**
5. **Model Definition** *LSTM with embedding and hidden layers for sentiment prediction*
6. **Model Training** use BCELoss or Binary Cross Entropy Loss to apply cross entropy loss to a single value between 0 and 1
7. **Model Performance Assessment** *on test data set*

6.2 Data Pre-Processing & Standardization

Data Cleaning

Lowercase all words, remove any punctuation, remove newline characters '\n' by splitting the text on '\n' and then join it back together

Tokenization

The text needs to be tokenized so that we have numerical data to feed into our model. Each word will be encoded as an integer (not a float) as we are using an embedding layer. We will not use the integer '0' as we want to reserve it to add some padding in the next step. Tokenization can be done from scratch or with built-in tokenizers of NLTK, spacy, etc. libraries.

Data Padding & Truncation

We want to create a set of movie reviews that are all of the same length. This means removing zero-length reviews as outliers and **Truncating** longer reviews (e.g. removing columns) and **Padding** shorter reviews with columns of zeroes.

7 Deploying PyTorch Models (Lesson 10)

7.1 Introduction

In production environments, models are typically required to run in C++. So, after development and training of a model in PyTorch, we could convert the models into a new representation called Torch Script which allows exporting models from Python into a C++ application (as of PyTorch 1.0). There are two methods for converting a model to Torch Script, **Tracing** and **Annotations**.

7.2 Converting to Torch Script with Tracing or Annotations

Tracing

We map out the structure of the model by passing an example tensor through it. We are basically doing a forward pass through the model with example data and have PyTorch keep track of all the operations performed on the inputs.

```
import torch
import torchvision

# An instance of your model.
model = torchvision.models.resnet18()

# An example input you would normally provide to your model's forward()
method.
example = torch.rand(1, 3, 224, 224)

# Use torch.jit.trace to generate a torch.jit.ScriptModule via tracing.
traced_script_module = torch.jit.trace(model, example)
```

We can then just pass in data into the traced_script_modul and it will output the result of a forward pass through the model.

```
In[1]: output = traced_script_module(torch.ones(1, 3, 224, 224))
In[2]: output[0, :5]
Out[2]: tensor([-0.2698, -0.0381,  0.4023, -0.3010, -0.0448], grad_fn=
<SliceBackward>)
```

Annotations

Some models have a control flow that doesn't work with the Tracing method. For example, there could be if statements in the forward method, which is fairly common in natural language processing problems.

Because the `forward` method of this module uses control flow that is dependent on the input, it is not suitable for tracing. Instead, we can convert it to a `ScriptModule`. In order to convert the module to the `ScriptModule`, one needs to compile the module with `torch.jit.script` as follows:

```
class MyModule(torch.nn.Module):
    def __init__(self, N, M):
        super(MyModule, self).__init__()
        self.weight = torch.nn.Parameter(torch.rand(N, M))

    def forward(self, input):
        if input.sum() > 0:
          output = self.weight.mv(input)
        else:
          output = self.weight + input
        return output

my_module = MyModule(10,20)
sm = torch.jit.script(my_module)
```

If you need to exclude some methods in your `nn.Module` because they use Python features that TorchScript doesn't support yet, you could annotate those with `@torch.jit.ignore`

`my_module` is an instance of `ScriptModule` that is ready for serialization.

7.3 Serialize the Script Module to a File

Once you have a `ScriptModule` in your hands, either from tracing or annotating a PyTorch model, you are ready to serialize it to a file. Later on, you'll be able to load the module from this file in C++ and execute it without any dependency on Python. Say we want to serialize the `ResNet18` model shown earlier in the tracing example. To perform this serialization, simply call save on the module and pass it a filename:

```
traced_script_module.save("traced_resnet_model.pt")
```

This will produce a `traced_resnet_model.pt` file in your working directory. If you also would like to serialize `my_module`, call `my_module.save("my_module_model.pt")` We have now officially left the realm of Python and are ready to cross over to the sphere of C++.

7.4 Load the Script Module in C++ with the C++ API

See the full tutorial on: https://pytorch.org/tutorials/advanced/cpp_export.html