Art'Em - 絵画風加工を VR で実現する: パート 5 (最終回)

この記事は、インテル® デベロッパー・ゾーンに公開されている「Art'Em – Artistic Style Transfer to Virtual Reality Final Update」の日本語参考訳です。

Art'Em は、コンピューター・ビジョンを使用して、バーチャル・リアリティー (VR) 互換の解像度で絵画風加工をリアルタイムに利用できるようにするアプリケーションです。このプログラムは、任意のソース (OpenCV* - ウェブカメラ、ユーザー画面、Android* フォンのカメラ (IP Webcam) など) からフィードを受け取り、スタイライズされたイメージを返します。

1. はじめに

このアプリケーションの開発には、さまざまなツールを使用しました。

Programs/ Frameworks	→ Windows / Ubuntu v17.04 → Python → Intel® Optimization for TensorFlow* → OpenCV → CUDA → PIL (Windows), Pyscreenshot (Linux) → Pygame/OpenCV → SciPy, NumPy → IP Webcam (For phone camera feed stylization)
Programming Languages	→ C++ → Python

*For training on the Intel® Nervana™ DevCloud

このシリーズは 3 つのセクションに分割することができ、それぞれのセクションでは絵画風加工プロセスを高速化する異なる方法について調査しました。

最初のセクションでは、XNOR-net の概念を紹介し、近似を使用せずに効率良く並列処理を実行する方法について詳細なケーススタディーを実施しました。この手法は有効ですが、カーネルを訓練可能なディープラーニング・モデルと統合することは、このプロジェクトの期間内には実現不可能なことが分かりました。

2 つ目のセクションでは、生成ネットワークと「ワンショット」のイメージ・スタイライゼーションの方法について調査しました。この手法は VR 互換の解像度でうまく動作しますが、各ネットワークが新しいスタイルを学習するのに長い時間がかかるという制限があります。

3 つ目のセクションでは、アダプティブ・インスタンスの正規化と呼ばれる手法を調査しました。この手法は、非常に高速にスタイライズできるだけでなく、任意のスタイルに瞬時に切り替えることが可能です。

2. XNOR ネットワーク

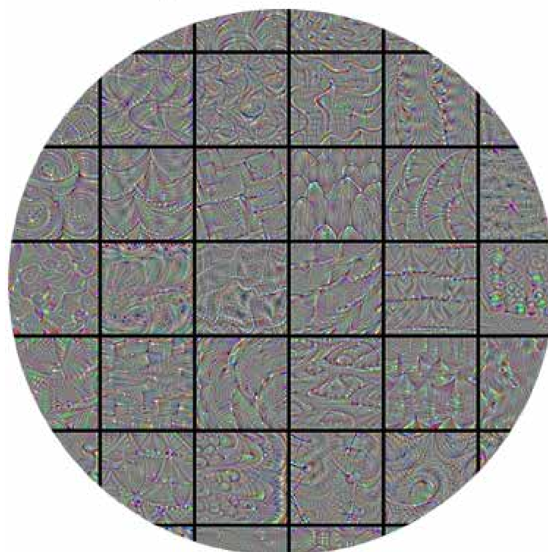
XNOR-net (否定排他的論理和) については、パート 1[1] と 2[2] で詳しく取り上げました。以下の図は、すべての行列要素が 1 または -1 の場合に、行列乗算操作を単純な操作 (XNOR と popcount) に置き換える方法を要約しています。これは、ネットワークのスピードアップが見込める可能性が高いことを示しています。

-1	+1	+1	-1	=	(-1x1) + (1x1) + (1x1)	=	3
-1	-1	-1	+1		(-1x-1) + (1x-1) + (1x-1)		-1
+1	+1	-1	+1		(-1x1) + (1x1) + (1x-1)		-1
0	+1	+1	0	=	popcnt(xnor(011,011))	=	3
0	0	0	+1		popcnt(xnor(011,000))		-1
+1	+1	0	+1		popcnt(xnor(011,110))		-1

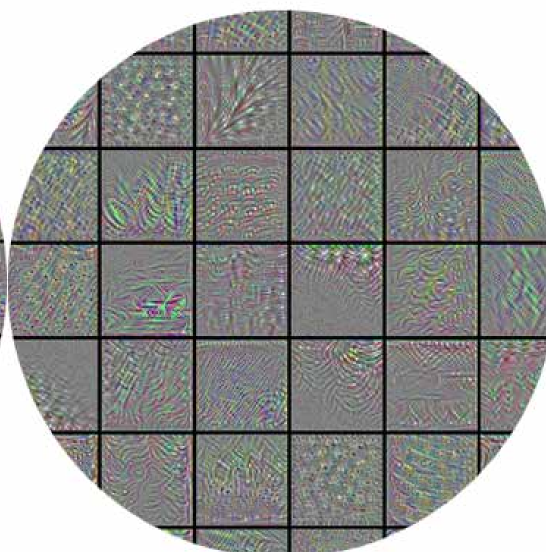
popcnt(xnor(011, 110)) = popcnt(xnor(0,1), xnor(1,1), xnor(1,0)) = popcnt(010) = -1 +1 -1 = -1

バイナリー化 (2 進化) の詳しい調査では、以下のイメージに示すように、イメージ·セマンティクスが大きく損なわれることが判明しました。しかし、これは非常に破壊的なバイナリー化手法の結果です。ネットワークの各行列をスカラー実数に +1 または -1 のみを含む行列を掛けたものに限定してネットワークを訓練すれば、適切な画像認識結果が得られることが証明[3] (英語) されました。AllenAI の結果では、訓練した AlexNet XNOR ネットワークが 43.3% という Top-1 精度を達成しました。

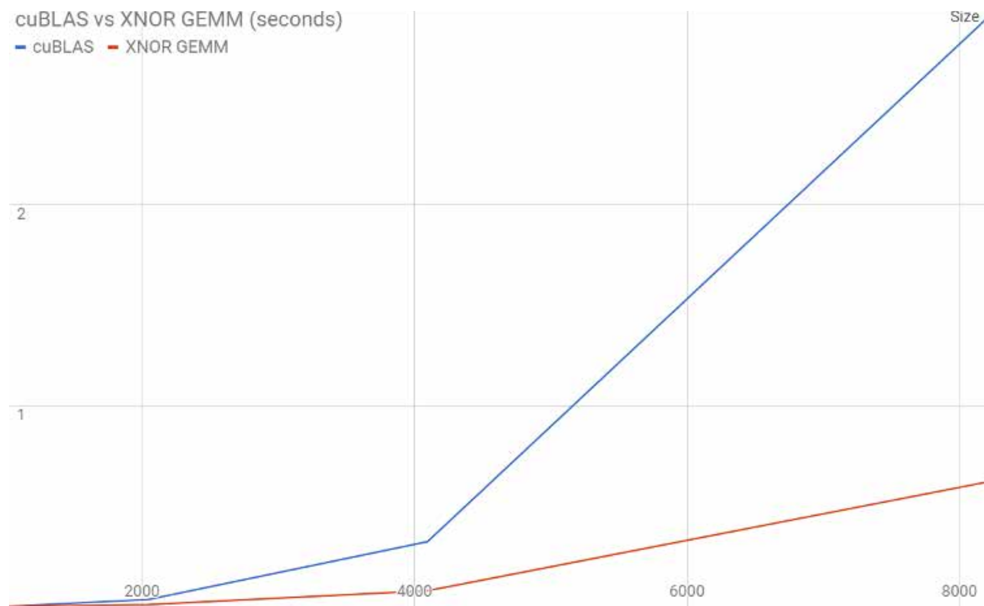
Full precision block 5 conv 1



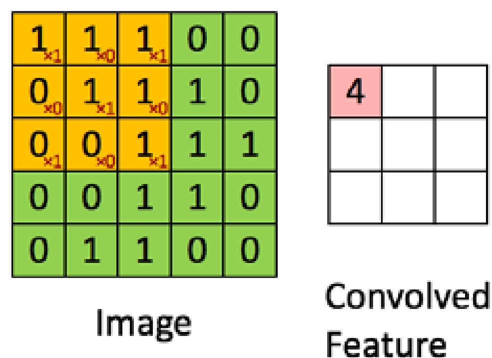
Binarized block 5 conv 1



パート 2 で説明したとおり、最適化されていない XNOR 汎用行列乗算カーネルを作成することで、cuBLAS と比較して最大 6 倍のスピードアップが得られます。これは、XNOR ネットワークが非常に限られたユースケースにのみ適用されるためであり、それほど驚くべきことではありません。



これは素晴らしいことですが、畳み込みニューラル・ネットワークにおける最重要部分は畳み込み操作です。畳み込み層におけるスピードアップは、全結合層ほど有望ではありません。両方で採用されている基本的な手法を実行すると、これが明らかになります。



[出典[4] (英語)]

上の図から分かるように、畳み込みでは、部分行列を選択してカーネルを掛ける必要があります。一方、行列乗算では、行列の行全体と列全体を掛けます。

XNOR-net で最も時間のかかるのは、ビットをデータ型にパックする処理です。MxM 行列を NxN 行列に畳み込む場合、$(N-M+1)^2$ 部分行列をデータ型にパックして、XNOR 関数と popcount を実行し、「畳み込んだ特徴」を生成します。

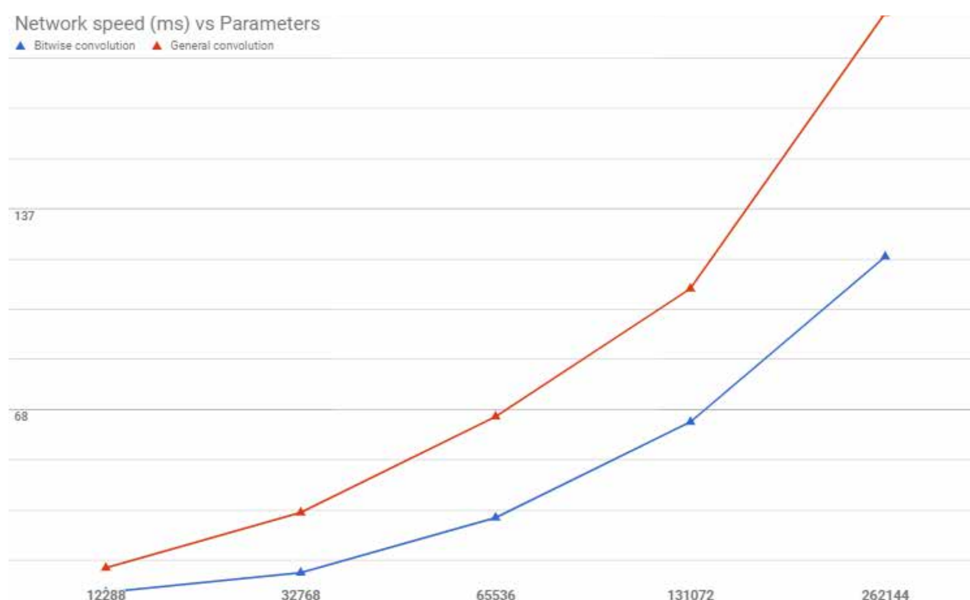
しかし、2 つの NxN 行列を掛け合わせる行列乗算では、2N 部分行列 (行と列) のみパックすればこの関数を実装できます。

つまり、低精度ネットワークで期待されるパフォーマンス・ゲインを下回ります。これを確認するため、CUDA (Compute Unified Device Architecture) C プログラミングを使用しました。実装は、こちらから入手[5] (英語) できます。このコードを実行するには、CUDA 互換のデバイスが必要です。異なるイメージサイズへの対応は、利用可能な VRAM に大きく依存します。ネットワークは、共有メモリーを利用して並列化されています。現在、4x4 カーネルでのみ動作しますが、簡単に拡張できます。

バイナリー化したカーネルの割り当ては、畳み込みタイマーが開始する前に行われます。ベンチマークは、nvprof ツールを使用して実行します。ネットワークのカーネルは、すでに適切な形式で保存されているため、カーネルをバイナリー化した後にアルゴリズムの時間を測定します。ブロックごとに長さ 256 の符号なし整数型の共有配列と、インデックス変数が割り当てられます。共有配列が共有メモリー内に割り当てられるように、ブロックサイズは (16,16) に設定します。そして、バイナリー化コードがカーネルの深さのループを開始して、入力「イメージ」の各チャネルを反復します。各反復において各スレッドは、配列の自身の要素にバイナリー化された部分行列をパックする符号なし整数を格納します。これが完了したら、カーネルと入力「イメージの」対応する部分行列を掛けた値を出力配列に格納するコード行を実行します。出力配列は平らですが、簡単に処理できます。TensorFlow* などのフレームワークにこれを適合するためチャネルを並列化する場合、最大グリッドサイズおよび最大ブロックサイズを超えないように細心の注意を払うことが重要です。そうしないと、メモリー割り当てとアクセスエラーが増加します。

これは、基本的な実装です。ネットワークは畳み込みごとに並列化されていますが、チャネル畳み込みは並列化されていません。以下に示すスピードアップの数倍を達成できる可能性がありますが、完全精度の畳み込み操作でも同じことが言えます。さらに並列化することでビット単位の畳み込みのほうが優れたパフォーマンスが得られますが、コードでは省略しました。

Image dimensions	Kernel dimensions	Parameters	Bitwise convolution (ms)	General convolution (ms)
256x256x3	4x4x256	12288	6.25	14.5
256x256x8	4x4x256	32768	12.865	33.324
256x256x16	4x4x256	65536	31.587	65.961
256x256x32	4x4x256	131072	64.14	109.52
256x256x64	4x4x256	262144	120.4	203.32



ネットワークは、基本的な汎用畳み込みカーネルよりも約 2 倍高速ですが、これは精度の低下を正当化するのに十分なスピードアップとは言えません。

バイナリー・ネットワークを使用する重要な利点の 1 つは、ネットワーク全体をはるかに小さな領域に格納できることです。つまり、VGG-19 モデル全体を約 512MB から約 16MB に減らすことができます。これは、大きな

モデルを保持することができない組込みデバイスで役立ちます。また、GPU や CPU でも、モデル全体を一度に VRAM にロードしたり、大きなデータセットを保持することができます。

このユースケースでは、大部分の精度を維持しつつ大幅なスピードアップをもたらす優れた手法を引き続き調査する必要があるでしょう。

3. 生成ネットワーク

こちらの論文[6] (英語) の研究を適合して、生成スタイライゼーション·ネットワークとインスタンス正規化の実装を調査しました。

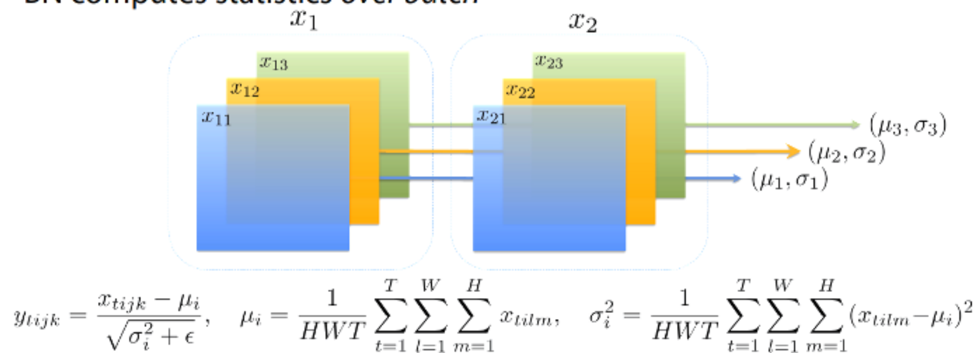
最適化ベースの絵画風加工モデルを実装することで、通常、より優れた、多様性のある結果が得られますが、かなり時間がかかります。適合するスタイルごとに生成ニューラル·ネットワークの学習が必要になりますが、最適化ベースの絵画風加工よりも画質と多様性を落とすことで、スタイライゼーションをスピードアップできます。

生成ネットワークの作成には 2 つのステップが含まれます。

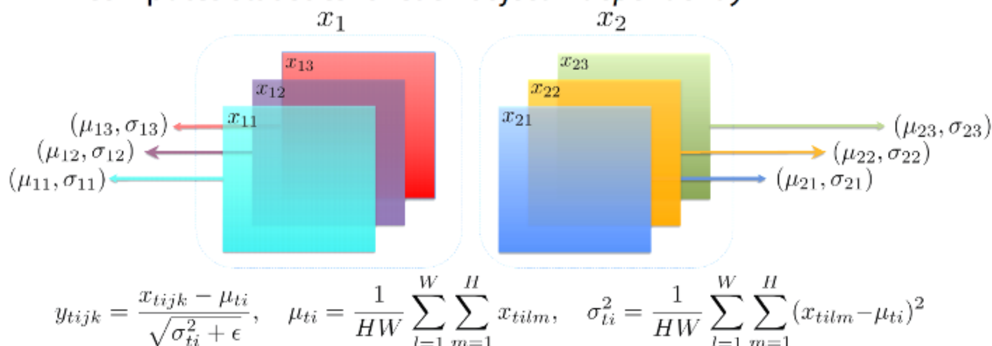
生成ネットワークの設計
これは、最も重要な部分です。生成ネットワークは、高品質な絵画風加工を提供するだけでなく、優れたフレームレートを達成するため小さくなければなりません。自身でネットワークを訓練しない場合、こちら[7] (英語) にある生成ネットワークの実装を使用することを推奨します。このネットワークは、3 つの畳み込み層、5 つの残差ブロック、2 つの転置畳み込み (畳み込み層で終わる) で構成されています。

生成ネットワークは、「インスタンスの正規化 (IN)」と呼ばれる正規化手法を採用しています。この手法は、バッチ正規化とは異なり、各バッチ要素の統計を個別に計算します。以下の図 (出典: Dmitry Ulyanov [8] (英語)) は、バッチ正規化とインスタンス正規化の相違点を示しています。IN はインスタンス正規化を、BN はバッチ正規化をそれぞれ表します。

• BN computes statistics *over batch*



• **IN** computes statistics *for each object independently*



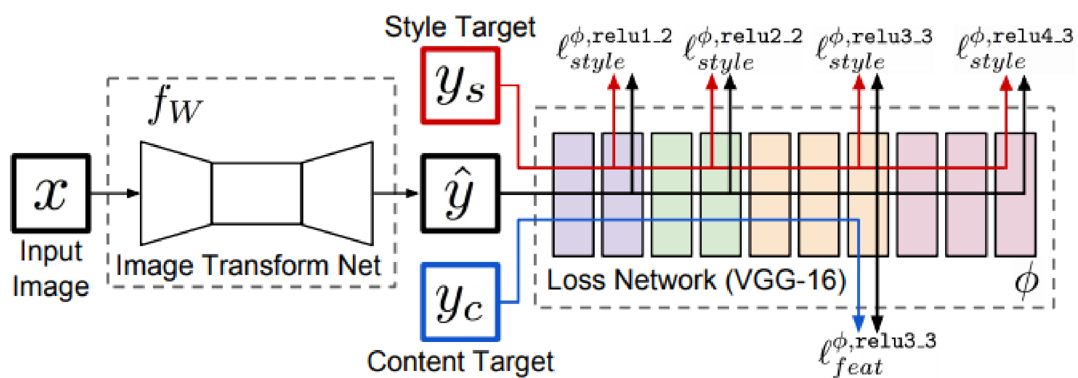
生成ネットワークの訓練
生成ネットワークをセットアップし、MS-COCO データセットと VGG19 モデルを保存したら、損失関数を実装する必要があります。

損失関数には、スタイルの損失、コンテンツの損失、全体的な変動の損失が含まれます。全体的な変動の損失を含めることで、生成されたイメージのノイズ除去に役立ちます。

入力イメージが生成ネットワークに読み込まれ、生成ネットワークの出力 (生成されたイメージと呼ぶ) がコンテンツターゲット、スタイルターゲットとともに VGG-19 モデルにフォワードされます。損失関数の値は、生成されたイメージ、スタイルターゲット、コンテンツターゲットを渡す際に VGG-19 の層を比較して計算されます。

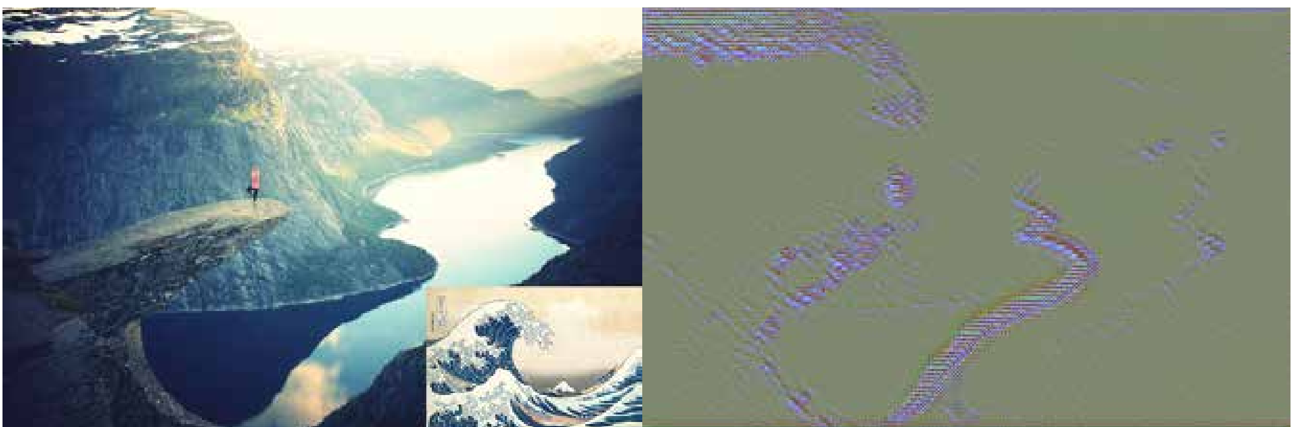
これには、ADAM オプティマイザーを利用しました。しかし、メモリー制限 BFGS 法 (L-BFGS) 最適化アルゴリズムのほうがはるかに良い結果をもたらし、最適化ベースの絵画風加工において畳み込みを高速化できました。

以下の図は、訓練プロセスを示しています。



[出典[9] (英語)]

以下の GIF 画像は、生成ネットワークの学習プロセスを示しています。学習プロセス全体はキャプチャーされていませんが、50 反復ごとにスナップショットが記録されています。インテル® Nervana™ DevCloud で訓練を行ったところ、わずか 900 反復で、イメージのスタイライズ方法について多くのことが学習されました。



訓練後、Image Transform Net でプロセス全体を実行できます。

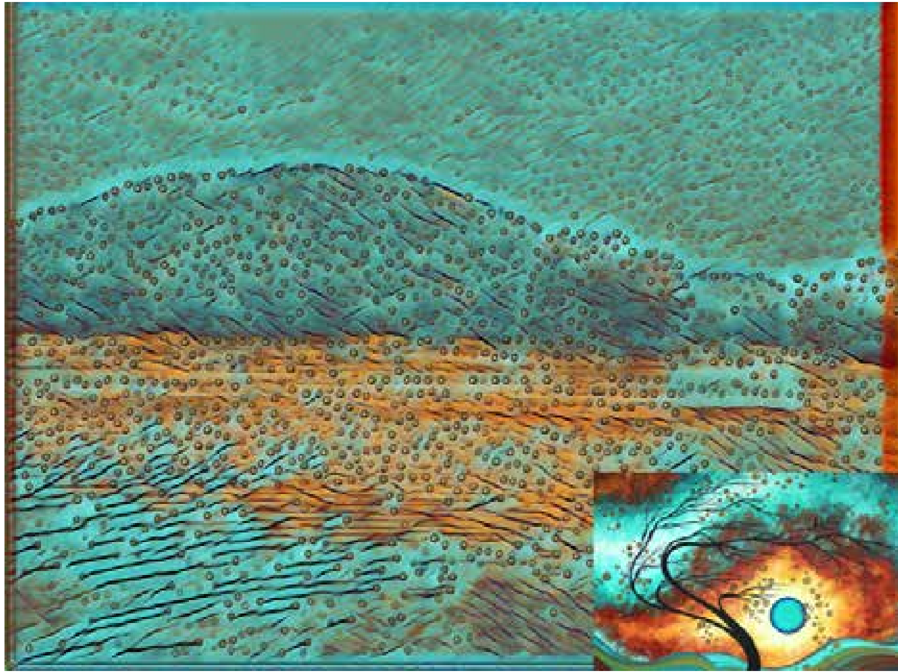
データセット

生成ネットワークの訓練には、MS-COCO データセット[10] (英語) を使用しました。MS-COCO データセットは、大規模なオブジェクト検出、セグメンテーション、キャプショニング・データセットです。ここでは、ラベルやオブジェクトの情報は不要で、ネットワークを訓練するため多数のイメージのみが必要です。これらがコンテンツイメージになります。スタイルイメージに関しては、生成ネットワークは 1 つのスタイルのみ学習できます。詳しいことは新しい手法とともに後述しますが、現在、ネットワークごとに 1 つのスタイルのみ訓練できます。

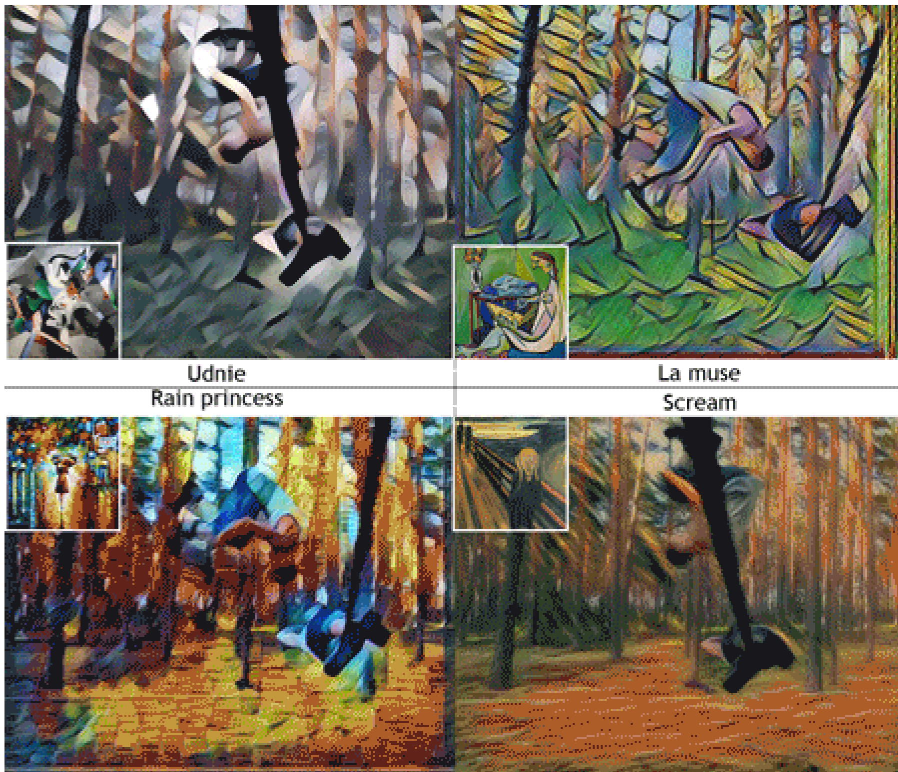
デフォルト設定の生成ネットワークを使用することで、解像度 600x540 で約 15fps を達成しました。そして、より小さな生成ネットワークを訓練したところ、フル VR 解像度で約 17fps を達成しました。このメトリックは、スタイライゼーションとともに ImageGrab 機能を使用した場合のものです。ImageGrab 機能は、デフォルトでは最大約 30fps を達成します。これは素晴らしい結果ですが、ネットワークを調整した結果、画質が大幅に低下しました。これらはすべて、ラップトップに搭載されているある程度強力なグラフィックス・プロセシング・ユニットで実行しました。したがって、デスクトップ・グレードの GTX 1080 Ti と強力なインテル® プロセッサーでは、フル VR 解像度で少なくとも 25fps を期待できます。

調整済みネットワークでのスタイライゼーションの結果は、あまり良いものではありませんでした。以下のイメージは、7000 反復後の結果です。訓練時間を長くすることで、より優れた結果を得られることが期待できます。一方、調整済みモデルのネットワーク速度は大幅に向上しました。

このネットワークはさらに訓練する予定です。右下のアーティファクトも改善が必要ですが、まだ効果的に訓練できていません。



以下の GIF 画像は、デフォルト設定のネットワークでスタイライゼーションを実行した結果です。VR の半分の解像度で、スタイライゼーションをリアルタイムに描画しています。同様の結果は、調整済みネットワークにおいて VR 解像度でも期待できます。



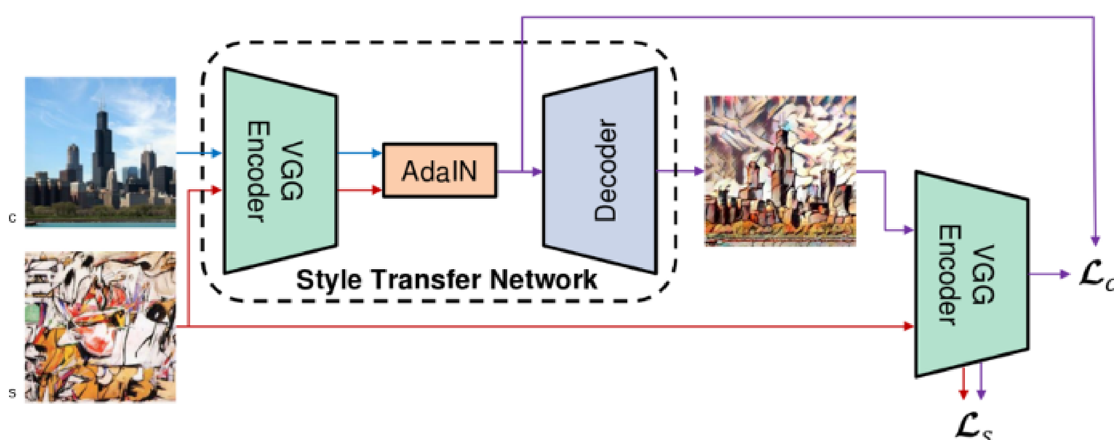
アプリケーションの現在の GUI は単純な Tkinter インターフェイスで、ImageGrab 機能を使って任意の解像度で画面をスタイライズできます。現在は概念の開発段階であるため、コードを透過的にすることに重点を置いており、VR インターフェイスへの適合が必要なフロントエンドの設計にはまだ取り掛かっていません。VR プラットフォームも重要であり、カメラ・フィード・ソースについても考慮する必要があります。現在のテスト用イ

ンターフェイスでは、5 つの事前に定義されたスタイライゼーション･オプションを提供しています。モデルは、こちらのリンク[11] (英語) からダウンロードできます。オリジナルのコードをダウンロードする場合は、こちら[12] (英語) のコードベースを使用して、簡単に自身のモデルを訓練できます。ウェブカメラ互換の OpenCV* モデルは、Android* の IP Webcam などのソフトウェアを利用して外部カメラ (電話) と統合しようと考えているため、まだ公開していません。

4. 複数のスタイルへの対応

上記の実装における重要な問題の 1 つは、新しいスタイルごとに新しいモデルの訓練が必要になることです。モデルの訓練には、高性能のハードウェアで約 4 ～ 6 時間かかります。ユーザーが好みに合わせて異なるスタイルを使用できる大規模なアプリケーションでは、これは明らかに受け入れられるものではありません。

高速なスタイライゼーション向けの以前のコードを維持しつつ、私は、アダプティブ･インスタンスの正規化手法に取り組んでみました。これは、どんなスタイルにも瞬時に適合でき、有望な結果をもたらす素晴らしい手法です。オリジナルの実装は、こちら[13] (英語) にあります。



$$\mathrm{AdaIN}(x, y) = \sigma(y)\left(\frac{x - \mu(x)}{\sigma(x)}\right) + \mu(y) \qquad \mathcal{L}_s = \sum_{i=1}^{L} \|\mu(\phi_i(g(t))) - \mu(\phi_i(s))\|_2 + \sum_{i=1}^{L} \|\sigma(\phi_i(g(t))) - \sigma(\phi_i(s))\|_2 \qquad \mathcal{L}_c = \|f(g(t)) - t\|_2$$

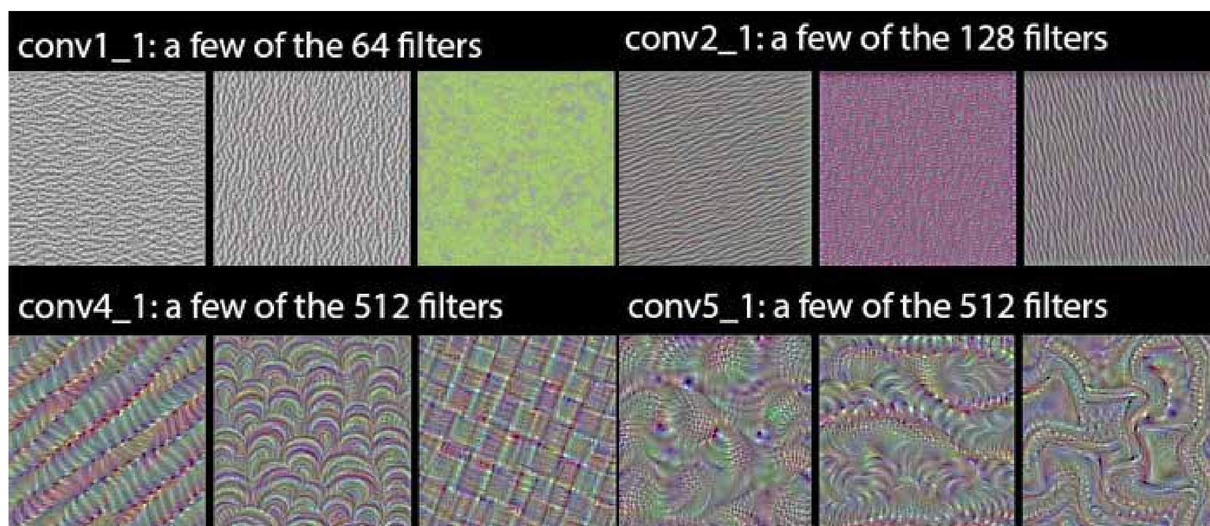
$\mu(x)$ Mean
$\sigma(y)$ Standard deviation
ϕ_i Layer in VGG19 used to compute style loss

t AdaIn(f(c), f(s))
f(t) VGG Encoder
g(t) Decoder

上記に示すアダプティブ･インスタンスの正規化手法の基本的な考え方は次のとおりです。

VGG エンコーダー

VGG-19 は、ImageNet モデルです。VGG-net などの画像認識モデルには、いくつかの畳み込み層と全結合層があります。全結合層は、畳み込み層のデータを任意の数のカテゴリーに「分解」します。ここでは、イメージの分類ではなく、イメージに関する情報を抽出することが目的のため、全結合層は必要ありません。VGG-net は優れた画像認識モデルであるため、各層にはイメージに関するデータが含まれていることでしょう。これらの層の出力は、以下のイメージから「想像する」ことができます。深さが増すほど、入力に関してより抽象的な情報が抽出されることが分かります。



つまり、全結合層を含まない VGG19 ネットワークを使用してイメージを渡す場合、VGG19 conv5_1 層の 512 のフィルターは、イメージに関する多くの重要な情報を抽出すると考えられます。

具体的には、この情報を利用して、「スタイル」情報を「コンテンツ」情報にエンコードできます。アダプティブ・インスタンスの正規化は、まさにこれを行います。

AdaIN

ネットワークの AdaIN モジュールは、コンテンツ「情報」x とスタイル「情報」y を受け取ります。そして、x のチャネル全体の平均と分散が y のものと一致するように調整します。これは訓練可能なモジュールではなく、単にスタイル情報に応じてコンテンツ情報を変換します。上記のモデルの全体図で、正確な関係を確認できます。

特定のスタイルの筆遣いを検出するスタイル「情報」から、特徴チャネルを想像することができます。これは非常に高い活性化を有し、入力コンテンツ「情報」x にスケーリングされます。これが、スタイライズされたエンコード済みのデータです。次に、この情報を実際のイメージへデコードする必要があります。

デコーダー

デコーダーの役割は、基本的に AdaIN モジュールの出力をデコードすることです。デコーダーはエンコーダーに似ています。プーリング層は、最も近いアップサンプリングで置き換えられます。デコーダーでは正規化は使用されません。これは、インスタンス正規化は、各サンプルを単一のスタイルに正規化し、バッチ正規化はサンプルのバッチに対して正規化を行うためです。

つまり、このネットワークは、エンコード済みデータに適切なスタイライゼーションを適用してオリジナルのサイズに戻します。

実装

オンラインの事前定義済みモデルを使用したことは効果的でしたが、よりコンパクトなネットワークを使用することで、ネットワーク速度を大幅に向上できたでしょう。コンテンツイメージ用には、MS-COCO データセットを使用し、スタイライゼーション・イメージには、デフォルトの Kaggle* データセット「Painter by numbers」を使用しました。このデータセットはダウンロード不可であったため、訓練には BAM (Behance-Artistic-Media) データセット[15] (英語) へのアクセスを申請して使用しました。

このモデルの実装時に初めて気付いたことは、アダプティブ・インスタンスの正規化モジュールが無効な場合、デコーダーは非常に低いコントラストのイメージを生成することでした。アート・データセットを入手するまでネットワークをさらに訓練することができなかったため、PIL イメージモジュールを利用して、デコーダー出力がより自然に、入力イメージに近くなるようにコントラストを向上させることにしました。エンコーダー・データを変換しない場合、デコーダーは単にオリジナルのイメージを返すため、この作業が必要でした。この後処理の影響から、ときにはややオーバーになることがあるものの、スタイライゼーション結果が格段に向上しました。

モデル全体を実装した後に、私は 2 つの重大な問題に気付きました。スタイライゼーションは標準的な結果になりましたが、色順応が良くありませんでした。これは、私の実装に問題がありました。チャネルのスタイル平均の重みをコンテンツ平均の重みよりも増やしたところ、色順応が大幅に向上しました。

事前に訓練されたモデルでは、研究論文で提示された画質を再現できませんでしたが、モバイル GTX 1070 において VR 解像度で約 8pfs を達成しました。より高性能なデスクトップ・グレードの VR 対応デバイスでは、さらなるスピードアップが期待できます。画質に関しては、大規模な特徴がデコーダーによって完全に変換されないことが原因である可能性もあります。より浅い VGG からの入力でデコーダーを訓練して、AdaIN モジュールを適用してみたいと考えています。これは、イメージに大きな変化が見られなかった理由の 1 つかもしれません。私のモデルでは、完全に変換されたスタイルはごくわずかです。

ネットワークのサイズをさらに減らして、VGG ネットワークの第 3 の畳み込みモジュールから入力を受け取るか、全く新しい ImageNet モデルを使用してデコーダーを作成することで、スタイライゼーションを大幅にスピードアップし、生成ネットワークを使用する前述のアプローチと同程度の速度が達成できる可能性があります。

以下は、この手法によるスタイライゼーションの結果です。ほとんどのイメージで、大きな変化が見られません。また、コントラストの増加によってわずかな色ずれが生じています。これは、今後、訓練によって改善されるでしょう。

まとめと今後の課題

このプロジェクトでは、リアルタイムの絵画風加工を実現し、カメラ/OpenCV* モジュールと統合しました。2 つのイメージのスタイライズ手法を調査し、どちらも有望な結果が得られました。このフレームワークは、VR 向けに簡単に拡張できます。また、XNOR-net の有用なパフォーマンス解析とその実行可能性も含まれています。

調整によりさらに向上できる点がいくつかあります。ラップトップ・グレードのグラフィカル・プロセシング・ユニットで 17 ～ 20 fps という快適なスタイライゼーション速度を達成する小さな生成ネットワークの訓練には、まだ取り掛かれていません。小さなエンコーダー/デコーダーペアを利用することで、ユーザーは任意のスタイルを適用できるようになります。また、AdaIn モジュールで使用する層を変更することで、より大規模な変化が見られると期待しています。アダプティブ・インスタンスの正規化の結果を改善するため、いくつかのパラメーターのチューニングも必要です。生成ネットワークは、妥当なフレームレートで優れた結果をもたらします。

謝辞

インテル® Nervana™ DevCloud で必要なトレーニング・リソースとこのアイデアを実現するためテクニカルサポートを提供してくれたインテル® Student Ambassador Program for AI に感謝します。このプロジェクトは、Intel Early Innovator Grant によって支援されました。

参考文献

[1] https://www.isus.jp/machine-learning/art-em-1/
[2] https://www.isus.jp/machine-learning/art-em-2/
[3] https://github.com/allenai/XNOR-Net (英語)
[4] https://cdn-images-1.medium.com/max/800/1*ZCjPUFrB6eHPRi4eyP6aaA.gif (英語)
[5] https://github.com/akhauriyash/XNOR-convolution/blob/master/xnorconv.cu (英語)
[6] https://arxiv.org/abs/1701.02096 (英語)
[7] https://github.com/lengstrom/fast-style-transfer/blob/master/src/transform.py (英語)
[8] https://dmitryulyanov.github.io/about (英語)
[9] https://cs.stanford.edu/people/jcjohns/papers/eccv16/JohnsonECCV16.pdf (英語)
[10] http://cocodataset.org/#home (英語)
[11] https://drive.google.com/drive/folders/0B9jhaT37ydSyRk9UX0wwX3BpMzQ?usp=sharing (英語)
[12] https://github.com/lengstrom/fast-style-transfer (英語)
[13] https://github.com/xunhuang1995/AdaIN-style (英語)
[14] https://github.com/jonrei/tf-AdaIN (英語)
[15] https://bam-dataset.org/ (英語)

前のステップ: パート 4

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。