نام و نام خانوادگی: حسنا اویارحسینی	گزارش پروژه درس سیستم های چندرسانه ای
شماره دانشجویی: ۹۸۲۳۰۱۰	تاریخ: خرداد - ۱۴۰۲

" فشرده سازی تصاویر "

بخش اول – سوالات تئوری)

۱- تفاوت فشرده سازی lossy, lossless و فرمت های هر یک:

فشرده سازی بدون اتلاف: فشرده سازی بدون اتلاف روشی برای کاهش حجم فایل در عین حفظ کامل داده های اصلی است. با حذف اطلاعات اضافی و یافتن الگوهای موجود در داده ها به فشرده سازی دست می یابد. هنگامی که یک فایل با استفاده از فشرده سازی بدون اتلاف فشرده می شود، می توان آن را بدون از دست دادن داده به شکل اصلی خود از حالت فشرده خارج کرد. الگوریتم فشرده سازی تضمین می کند که داده های اصلی می توانند به طور کامل بازسازی شوند. نمونه هایی از فرمت های تصویر بدون اتلاف عبارتند از PNG (گرافیک شبکه قابل حمل)، GIF (فرمت تبادل گرافیکی) و TIFF (فرمت فایل تصویر برچسب گذاری شده). فشرده سازی با اتلاف: فشرده سازی با اتلاف روشی است که برخی از داده ها را برای دستیابی به نسبت تراکم بالاتر قربانی می کند. با حذف اطلاعات غیر ضروری یا کمتر قابل توجه از فایل به این امر دست می یابد. هدف فشرده سازی Lossy کاهش قابل توجه اندازه فایل در حالی که سطح قابل قبولی از کیفیت درک شده را حفظ می کند. فایل از حالت فشرده خارج شده با فایل اصلی یکسان نخواهد بود، زیرا برخی از داده ها برای همیشه کنار گذاشته شده اند. فرمت‌های متداول تصویر با اتلاف عبارتند از JPEG (گروه مشترک متخصصان عکاسی) و WebP (فرمت تصویر وب).

۲- گام های فشرده سازی JPEG:

شامل چندین مرحله است، از جمله تبدیل کسینوس گسسته (DCT)، کوانتیزاسیون و کدگذاری. در اینجا توضیحی در مورد هر مرحله آورده شده است:

- تبدیل فرمت تصویر:

ابتدا تصویر RGB را به YIQ یا YUV تبدیل میکنیم زیرا میزان روشنایی از نظر چشم انسان مهم تر است و در این دو فرمت روشنایی به عنوان یک مولفه لحاظ شده.

- تبدیل کسینوس گسسته (DCT):

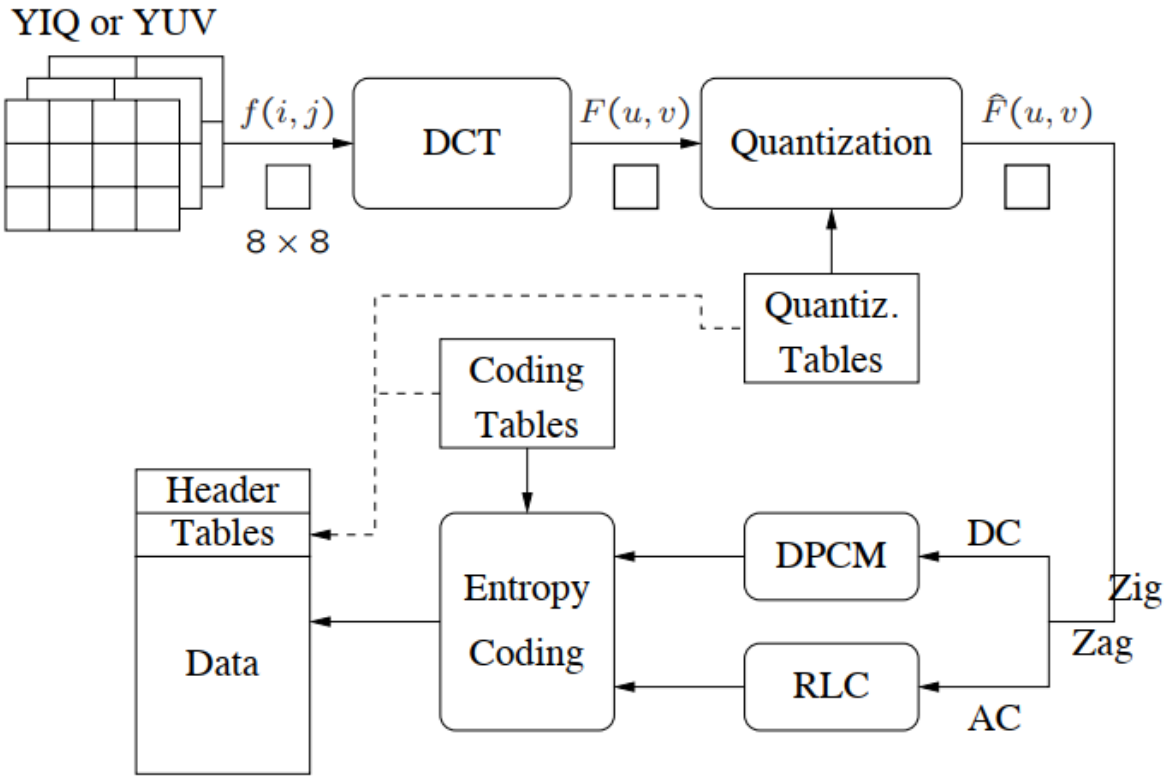
DCT اولین گام در فشرده سازی JPEG است. تصویر به بلوک های ۸×۸ پیکسل تقسیم می شود و DCT (تبدیل کسینوس گسسته) به طور مستقل برای هر بلوک اعمال می شود در واقع داده های تصویر را از حوزه فضایی به حوزه فرکانس تبدیل می کند. DCT تصویر را به یک سری اجزای فرکانس به نام ضرایب DCT تجزیه می کند. این ضرایب نشان دهنده فرکانس های مختلف موجود در تصویر است. DCT به متمرکز کردن اطلاعات تصویر در تعداد کمتری از ضرایب کمک می کند و امکان فشرده سازی موثرتر را فراهم می کند.

- کوانتیزاسیون:

پس از انجام DCT، ضرایب DCT حاصل کوانتیزه می شوند. کوانتیزاسیون شامل تقسیم هر ضریب بر مجموعه ای از مقادیر کوانتیزاسیون است. هدف کوانتیزاسیون کاهش دقت ضرایب و تعداد بیت لازم است. این مرحله امکان کاهش قابل توجهی در مقدار داده های مورد نیاز برای نمایش تصویر را فراهم می کند. فرآیند کوانتیزاسیون باعث از دست رفتن اطلاعات می شود، زیرا برخی از ضرایب گرد یا کوتاه شده اند. این از دست دادن اطلاعات باعث برگشت ناپذیری فشرده سازی JPEG می شود.

- کدگذاری:

در مرحله کدگذاری، ضرایب DCT کوانتیزه شده با استفاده از کدگذاری با طول متغیر بیشتر فشرده می‌شوند. ابتدا ضرایب DC به روش DPCM و ضرایب AC به صورت RLC کد میشود سپس مقادیر کد شده با entropy coding کد میشوند. متداول ترین روش مورد استفاده در فشرده سازی JPEG، کدگذاری هافمن است. کدگذاری هافمن کدهای کوتاه تری را به ضرایب متداول و کدهای طولانی تر را به ضرایب کمتر اختصاص می دهد. این فرآیند حجم کلی فایل فشرده را کاهش می دهد.



۳- چهار حالت ذخیره سازی با فرمت JPEG:

- حالت متوالی:

حالت ترتیبی ساده ترین و رایج ترین حالت در فشرده سازی JPEG است. در این حالت، تصویر به بلوک های ۸×۸ پیکسل تقسیم می شود و DCT (تبدیل کسینوس گسسته) به طور مستقل برای هر بلوک اعمال می شود. سپس ضرایب DCT حاصل کوانتیزه می شوند، با استفاده از کدگذاری هافمن کدگذاری می شوند و به ترتیب متوالی ذخیره می شوند. این حالت تصویر را به تدریج فشرده می کند، به این معنی که داده های فشرده شده از چپ به راست و از بالا به پایین کدگذاری می شوند. نسبت فشرده سازی خوبی را فراهم می کند اما امکان رندر پیش رونده تصویر را نمی دهد.

- حالت پیشرو:

حالت پیشرونده در فشرده‌سازی JPEG راهی را برای نمایش سریع یک نسخه با کیفیت پایین از تصویر فراهم می‌کند و با دریافت داده‌های بیشتر، کیفیت به تدریج بهبود می‌یابد. این کار با رمزگذاری تصویر در چند پاس انجام می‌شود و هر پاس جزئیات تصویر را اصلاح می‌کند. در حالت پیشرونده، ضرایب DCT به چندین باند فرکانسی تقسیم می‌شوند که از جزء DC (جریان مستقیم) شروع می‌شود که میانگین شدت رنگ هر بلوک را نشان می‌دهد. سپس ضرایب درون هر باند کوانتیزه می‌شوند، کدگذاری می‌شوند و در اسکن‌های متوالی ارسال می‌شوند. این حالت امکان پیش نمایش تصویری با کیفیت پایین تر را قبل از رندر کامل تصویر فراهم می کند.

حالت سلسله مراتبی:

حالت سلسله مراتبی، همچنین به عنوان "رمزگذاری هرمی" شناخته می شود، برای پشتیبانی از فشرده سازی مقیاس پذیر و انتقال کارآمد تصاویر طراحی شده است. این تصویر را در سطوح یا لایه‌های مختلف با وضوح کدگذاری می‌کند و امکان انتقال و رمزگشایی تدریجی را فراهم می‌کند. در حالت سلسله مراتبی، تصویر به سطوح مختلف تفکیک پذیری مانند لایه های با وضوح پایین، متوسط و وضوح بالا تقسیم می شود. هر لایه به طور جداگانه با استفاده از فشرده سازی JPEG فشرده می شود و از پایین ترین وضوح شروع می شود. این حالت رمزگشایی انتخابی سطوح وضوح خاص را امکان پذیر می کند و انعطاف پذیری در نمایش تصویر بر اساس منابع موجود و ترجیحات کاربر را فراهم می کند.

حالت losslss:

برخلاف سه حالت قبلی، حالت بدون اتلاف در فشرده سازی JPEG تمام داده های تصویر اصلی را بدون از دست دادن اطلاعات حفظ می کند. با استفاده از تکنیک های کدگذاری پیش بینی و کدگذاری آنتروپی به فشرده سازی دست می یابد. در حالت بدون اتلاف، تصویر به واحدهای پیش بینی تقسیم می شود و تفاوت بین مقادیر پیکسل پیش بینی شده و واقعی کدگذاری می شود. سپس داده های رمزگذاری شده با استفاده از کدگذاری آنتروپی، مانند هافمن یا کدگذاری حسابی، فشرده می شوند. حالت Lossless تضمین می کند که تصویر اصلی می تواند دقیقاً از داده های فشرده بازسازی شود. با این حال، نسبت تراکم به دست آمده در حالت بدون تلفات به طور کلی در مقایسه با حالت های اتلاف کمتر است.

٤- مزایا و معایب فشرده سازی هافمن:

مزایای فشرده سازی هافمن:

- فشرده‌سازی کارآمد: کدگذاری هافمن با اختصاص کدهای کوتاه‌تر به نمادها یا الگوهای داده‌ای که بیشتر اتفاق می‌افتند و کدهای طولانی‌تر به نمادهای کم‌تکرار، فشرده‌سازی کارآمد را به دست می‌آورد. این منجر به نمایش کد با طول متغیر می شود که طول کلی رمزگذاری را بهینه می کند. فشرده سازی هافمن به ویژه در فشرده سازی داده ها با توزیع فرکانس ناهموار موثر است، زیرا می تواند به طور قابل توجهی میانگین تعداد بیت های مورد نیاز برای نمایش داده ها را کاهش دهد.

- فشرده سازی بدون تلفات: کدگذاری هافمن یک روش فشرده سازی بدون تلفات است، به این معنی که داده های اصلی را می توان به طور کامل از داده های فشرده بازسازی کرد. هیچ گونه از دست دادن یا تخریب در داده ها در طول فشرده سازی ایجاد نمی کند. این باعث می‌شود فشرده‌سازی هافمن برای سناریوهایی که حفظ یکپارچگی داده‌ها ضروری است، مانند فایل‌های متن یا برنامه، مناسب باشد.

- سادگی و سرعت: اجرای کدگذاری هافمن از نظر سرعت رمزگذاری و رمزگشایی نسبتاً ساده و کارآمد است. عملیات الگوریتم شامل ساخت درخت هافمن است که می تواند با یک فرآیند ساده انجام شود و مراحل رمزگذاری و رمزگشایی جستجوهای ساده جدول هستند. این سادگی به کارایی و کاربردی بودن فشرده سازی هافمن کمک می کند.

معایب فشرده سازی هافمن:

- عدم بهینه سازی جهانی: کدگذاری هافمن طول کد را بر اساس فرکانس های نماد محلی بهینه می کند. با این حال، وابستگی یا همبستگی عمومی و کلی بین نمادها را در نظر نمی گیرد. در نتیجه، ممکن است در مواردی که چنین وابستگی هایی وجود دارد، به بهترین فشرده سازی ممکن دست پیدا نکند. سایر تکنیک‌های فشرده‌سازی، مانند کدگذاری محاسباتی، می‌توانند فشرده‌سازی بهتری را در شرایطی که بهینه جهانی بسیار مهم است، فراهم کنند.

- طول کد اضافی: کدگذاری هافمن از طول کدهای عدد صحیح برای نشان دادن نمادها استفاده می کند، که ممکن است هنگام نمایش نمادهایی با فرکانس های مشابه منجر به افزونگی شود. در برخی موارد، طول کد می‌تواند مهم‌تر از حد لازم باشد که منجر به نسبت‌های فشرده‌سازی کارآمدتر می‌شود. سایر تکنیک‌های کدنویسی، مانند رمزگذاری حسابی، می‌توانند با استفاده از طول کد کسری، این مسئله را بهتر کنترل کنند.

- عدم دسترسی تصادفی: کدگذاری هافمن یک کد بدون پیشوند تولید می کند، به این معنی که هیچ کدی پیشوند کد دیگری نیست. در حالی که این ویژگی برای کارایی رمزگشایی مفید است، فاقد قابلیت دسترسی تصادفی است. برای دسترسی به یک نماد یا بخش داده خاص، کل داده های فشرده شده باید از ابتدا رمزگشایی شوند، که ممکن است در برنامه های خاصی که دسترسی تصادفی به عناصر داده خاص مورد نیاز است، مطلوب نباشد.

- درخت هافمن: برای دیکود کردن نیاز به انتقال درخت هافمن داریم که حجم زیادی دارد

۵- فشرده سازی تصویر و اهیمت آن:

فشرده‌سازی تصویر به فرآیند کاهش اندازه تصاویر دیجیتالی در حین تلاش برای حفظ کیفیت بصری و محتوای آنها اشاره دارد. این یک تکنیک حیاتی در زمینه های مختلف از جمله عکاسی، طراحی گرافیک، چند رسانه ای و توسعه وب است. در اینجا برخی از جنبه های کلیدی و اهمیت فشرده سازی تصویر آورده شده است:

- کاهش حجم فایل: فشرده سازی تصویر به میزان قابل توجهی حجم فایل تصاویر را کاهش می دهد. این برای ذخیره سازی و انتقال کارآمد تصاویر بسیار مهم است، به ویژه در سناریوهایی با ظرفیت ذخیره سازی محدود یا محدودیت های پهنای باند. اندازه فایل های کوچکتر بارگذاری سریعتر تصویر در وب سایت ها، انتقال سریعتر از طریق شبکه و استفاده کارآمد از منابع ذخیره سازی را امکان پذیر می کند.

- انتقال سریعتر داده ها: تصاویر فشرده به زمان کمتری برای انتقال از طریق شبکه نیاز دارند و باعث می شود وب سایت های پر تصویر سریعتر بارگذاری شوند. این امر به ویژه در چشم انداز دیجیتال امروزی که کاربران انتظار دارند صفحات وب با سرعت بارگذاری سریع و تجربه کاربری بهینه را دارند، اهمیت ویژه ای دارد. با کاهش اندازه فایل تصویر، فشرده سازی تصویر به بهبود عملکرد وب سایت و کاهش هزینه های انتقال داده کمک می کند.

- بهینه سازی پهنای باند: فشرده سازی تصویر نقش مهمی در حفظ پهنای باند ایفا می کند، به خصوص در شرایطی که منابع شبکه محدود است. با کاهش اندازه تصاویر، داده های کمتری باید منتقل شود و در نتیجه مصرف پهنای باند کاهش می یابد. این امر به ویژه در زمینه هایی مانند شبکه های تلفن همراه، که در آن پهنای باند اغلب محدود یا گران است، مرتبط است.

- کارایی ذخیره‌سازی: تصاویر فشرده فضای ذخیره‌سازی کمتری را اشغال می‌کنند و امکان استفاده کارآمدتر از منابع ذخیره‌سازی را فراهم می‌کنند. این امر به‌ویژه در محیط‌هایی که مجموعه‌های بزرگی از تصاویر باید ذخیره شوند، مانند آرشیوها، پایگاه‌های اطلاعاتی یا پلتفرم‌های مبتنی بر ابر، مرتبط است. فشرده سازی تصویر باعث صرفه جویی در هزینه و مدیریت کارآمد زیرساخت های ذخیره سازی می شود.

- اشتراک گذاری و توزیع تصویر: با گسترش رسانه های اجتماعی، پلتفرم های پیام رسانی و برنامه های کاربردی تصویر محور، فشرده سازی تصویر برای تسهیل اشتراک گذاری و توزیع آسان تصاویر بسیار مهم است. تصاویر فشرده شده را می توان به سرعت آپلود، دانلود و در پلتفرم های مختلف به اشتراک گذاشت و دسترسی و تعامل کاربر را افزایش داد.

- حفظ کیفیت بصری: در حالی که فشرده سازی تصویر حجم فایل را کاهش می دهد، هدف آن حفظ سطح قابل قبولی از کیفیت بصری است. تکنیک های مختلف فشرده سازی، مانند فشرده سازی با اتلاف و بدون اتلاف، برای ایجاد تعادل بین کاهش اندازه فایل و کیفیت بصری استفاده می شود. حفظ جزئیات بصری ضروری تصاویر برای اطمینان از اینکه آنها از نظر بصری جذاب باقی می مانند و به هدف مورد نظر خود عمل می کنند بسیار مهم است.

٦- تفاوت فشرده سازی spatial, transform based. مزایا و معایب هر یک:

فشرده سازی فضایی:

فشرده سازی فضایی که به عنوان فشرده سازی مستقیم نیز شناخته می شود، مستقیماً بر روی مقادیر پیکسل یک تصویر عمل می کند. این شامل استفاده از تکنیک های مختلف برای کاهش افزونگی در داده های تصویر است. تکنیک‌های فشرده‌سازی فضایی شامل رمزگذاری طول اجرا، کدگذاری هافمن و کدگذاری پیش‌بینی است. در اینجا مزایا و معایب فشرده سازی فضایی وجود دارد:

- مزایای:

سادگی: تکنیک‌های فشرده‌سازی فضایی در مقایسه با روش‌های فشرده‌سازی مبتنی بر تبدیل، برای پیاده‌سازی و درک نسبتاً ساده هستند.

رمزگذاری و رمزگشایی سریع: الگوریتم‌های فشرده‌سازی فضایی اغلب فرآیندهای رمزگذاری و رمزگشایی سریع‌تری دارند، زیرا آنها مستقیماً مقادیر پیکسل را بدون تبدیل‌های پیچیده ریاضی دستکاری می‌کنند.

مناسب برای تصاویر ساده: فشرده سازی فضایی برای تصاویر با پیچیدگی نسبتاً کم یا زمانی که الزامات فشرده سازی سختگیرانه نیست، به خوبی کار می کند.

- معایب:

نسبت فشرده سازی محدود: روش های فشرده سازی فضایی ممکن است به نسبت تراکم بالایی در مقایسه با تکنیک های مبتنی بر تبدیل دست پیدا نکنند. آنها ممکن است ساختارهای پیچیده تصویر را به طور کارآمد ثبت نکنند یا همه اشکال افزونگی را حذف نکنند.

چالش‌های فشرده‌سازی با اتلاف: تکنیک‌های فشرده‌سازی فضایی ممکن است برای رسیدن به فشرده‌سازی بالا با حفظ کیفیت تصویر خوب مشکل داشته باشند. نسبت فشرده سازی بالا در فشرده سازی فضایی اغلب منجر به از دست دادن قابل توجه جزئیات تصویر و مصنوعات بصری می شود.

فشرده سازی مبتنی بر تبدیل:

فشرده‌سازی مبتنی بر تبدیل شامل تبدیل داده‌های تصویر از حوزه فضایی به حوزه دیگری (مانند دامنه فرکانس یا موجک) با استفاده از تبدیل‌های ریاضی مانند تبدیل کسینوس گسسته (DCT) یا تبدیل موجک گسسته (DWT) است. این تبدیل به تمرکز انرژی تصویر در ضرایب تبدیل کمتر کمک می کند و امکان فشرده سازی کارآمدتر را فراهم می کند. در اینجا مزایا و معایب فشرده سازی مبتنی بر تبدیل آورده شده است:

- مزایا:

نسبت تراکم بالاتر: تکنیک های فشرده سازی مبتنی بر تبدیل می توانند نسبت تراکم بالاتری را در مقایسه با فشرده سازی فضایی به دست آورند. آنها از افزونگی ذاتی در ضرایب تبدیل شده برای دستیابی به راندمان فشرده سازی بهتر استفاده می کنند.

حفظ بهتر کیفیت تصویر: روش‌های فشرده‌سازی مبتنی بر تبدیل، به‌ویژه آنهایی که انواع بدون تلفات دارند، می‌توانند کیفیت تصویر را در مقایسه با فشرده‌سازی فضایی حفظ کنند. آنها می توانند جزئیات بیشتری را در حین دستیابی به نسبت تراکم خوب حفظ کنند.

- معایب:

پیچیدگی: روش‌های فشرده‌سازی مبتنی بر تبدیل عموماً برای پیاده‌سازی پیچیده‌تر هستند و به عملیات ریاضی پیچیده‌ای برای تبدیل و فشرده‌سازی نیاز دارند. این پیچیدگی می تواند بر سرعت رمزگذاری و رمزگشایی تأثیر بگذارد.

الزامات محاسباتی: تکنیک های فشرده سازی مبتنی بر تبدیل، مانند DCT یا DWT، به منابع محاسباتی قابل توجهی برای رمزگذاری و رمزگشایی نیاز دارند. این می تواند یک محدودیت در سناریوهایی با قدرت پردازش محدود یا نیازهای بلادرنگ باشد.

مبتنی بر بلوک: برخی از روش‌های فشرده‌سازی مبتنی بر تبدیل، مانند JPEG مبتنی بر DCT، بر روی بلوک‌های با اندازه ثابت یک تصویر عمل می‌کنند که می‌تواند مصنوعات مبتنی بر بلوک را در تصویر غیرفشرده‌شده معرفی کند.

۷- استفاده از فشرده سازی تصویر در دنیای واقعی

فشرده سازی تصویر به طور گسترده در برنامه های مختلف دنیای واقعی برای بهینه سازی ذخیره سازی، انتقال و نمایش تصاویر استفاده می شود. در اینجا چند نمونه از نحوه استفاده از فشرده سازی تصویر آورده شده است:

- گرافیک وب: وب سایت ها اغلب از تکنیک های فشرده سازی تصویر برای بهینه سازی سرعت بارگذاری و عملکرد کلی استفاده می کنند. تصاویر فشرده حجم فایل را کاهش می دهند، به صفحات وب اجازه می دهند سریعتر بارگذاری شوند، تجربه کاربر را بهبود می بخشند و مصرف پهنای باند را کاهش می دهند. فرمت هایی مانند JPEG و WebP معمولاً برای فشرده سازی با اتلاف گرافیک های وب استفاده می شوند.

- عکاسی دیجیتال: فشرده سازی تصویر نقشی حیاتی در عکاسی دیجیتال دارد. عکاسان حرفه ای اغلب تصاویر را در فرمت های خام، بزرگ و غیر فشرده می گیرند تا حداکثر کیفیت تصویر را حفظ کنند. با این حال، برای اهداف اشتراک‌گذاری، انتشار یا ذخیره‌سازی، این تصاویر با استفاده از فرمت‌هایی مانند JPEG فشرده می‌شوند تا اندازه فایل کاهش یابد و در عین حال سطح قابل قبولی از کیفیت بصری حفظ شود.

- پخش ویدئو: فشرده سازی ویدئو به شدت به تکنیک های فشرده سازی تصویر متکی است. در سرویس‌های پخش ویدیو، مانند YouTube یا Netflix، محتوای ویدیویی با استفاده از فرمت‌هایی مانند .H۲۶۴ یا .H۲۶۵ (HEVC) فشرده می‌شود. این فرمت‌ها از روش‌های فشرده‌سازی تصویر برای کاهش اندازه فریم‌های ویدئویی استفاده می‌کنند و امکان پخش و پخش کارآمد در دستگاه‌های مختلف با پهنای باند محدود را فراهم می‌کنند.

- برنامه‌های کاربردی موبایل: فشرده‌سازی تصویر برای برنامه‌های تلفن همراه، به‌ویژه برنامه‌هایی که شامل اشتراک‌گذاری تصویر یا پلت‌فرم‌های رسانه‌های اجتماعی هستند، بسیار مهم است. دستگاه های تلفن همراه ظرفیت ذخیره سازی محدودی دارند و اغلب با اتصالات شبکه کندتر کار می کنند. فرمت‌های تصویر فشرده، مانند JPEG، به کاربران اجازه می‌دهند تا تصاویر را به سرعت و کارآمد بارگذاری، دانلود و به اشتراک بگذارند و در عین حال مصرف داده را به حداقل می‌رسانند.

- تصویربرداری پزشکی: در تصویربرداری پزشکی، جایی که تصاویر با وضوح بالا و جزئیات برای تشخیص و تجزیه و تحلیل بسیار مهم هستند، معمولاً از تکنیک‌های فشرده‌سازی بدون تلفات استفاده می‌شود. فرمت‌هایی مانند DICOM (تصویربرداری دیجیتال و ارتباطات در پزشکی) از فشرده‌سازی بدون تلفات برای کاهش نیازهای ذخیره‌سازی استفاده می‌کنند و در عین حال اطمینان می‌دهند که تصاویر پزشکی می‌توانند با دقت و بدون از دست دادن اطلاعات بازسازی شوند.

- تصاویر ماهواره‌ای: تصاویر ماهواره‌ای که برای نقشه‌برداری، سنجش از دور یا نظارت بر محیط استفاده می‌شوند، معمولاً اندازه بزرگی دارند. فشرده سازی تصویر برای کاهش اندازه فایل این تصاویر برای ذخیره سازی، انتقال و تجزیه و تحلیل استفاده می شود. تکنیک‌های فشرده‌سازی مانند فشرده‌سازی مبتنی بر موجک، مانند JPEG ۲۰۰۰، معمولاً برای حفظ جزئیات تصویر و اطمینان از مدیریت کارآمد مجموعه داده‌های تصویر ماهواره‌ای بزرگ استفاده می‌شوند.

۸- مشکلات رایج فشرده سازی تصویر و روش دوری از آن

- از دست دادن کیفیت تصویر: تکنیک های فشرده سازی از دست رفته، مانند JPEG، می تواند منجر به کاهش کیفیت تصویر شود. برای جلوگیری از تخریب بیش از حد، مهم است که پارامترهای فشرده سازی را با دقت انتخاب کنیم، مانند نسبت تراکم، جداول کوانتیزاسیون و نمونه برداری رنگی. استفاده از نسبت فشرده سازی کمتر و تنظیمات با کیفیت بالاتر می تواند به به حداقل رساندن مصنوعات قابل مشاهده و حفظ جزئیات تصویر کمک کند.

- پیچیدگی محاسباتی: برخی از تکنیک های فشرده سازی تصویر، به ویژه آنهایی که مبتنی بر تبدیل ها یا الگوریتم های پیشرفته هستند، می توانند محاسباتی فشرده باشند. این ممکن است در برنامه های بلادرنگ یا در دستگاه هایی با قدرت پردازش محدود چالش هایی ایجاد کند. برای پرداختن به این موضوع، تکنیک های بهینه سازی، مانند پردازش موازی، شتاب سخت افزاری، یا استراتژی های فشرده سازی تطبیقی، می توانند برای کاهش پیچیدگی محاسباتی به کار گرفته شوند.

- مشکلات سازگاری: فرمت های مختلف فشرده سازی تصویر ممکن است به طور جهانی توسط همه دستگاه ها یا برنامه های نرم افزاری پشتیبانی نشوند. این می تواند منجر به مشکلات سازگاری هنگام اشتراک گذاری یا نمایش تصاویر فشرده شود. برای کاهش مشکلات سازگاری، توصیه می‌شود فرمت‌هایی را انتخاب کنیم که به طور گسترده پشتیبانی می‌شوند، از الگوریتم‌های فشرده‌سازی استاندارد استفاده کنید، و هنگام اشتراک‌گذاری یا توزیع تصاویر، گزینه‌های بازگشتی یا قالب‌های جایگزین ارائه کنیم.

- معاوضه بین نسبت فشرده سازی و کیفیت تصویر: غالباً بین دستیابی به نسبت فشرده سازی بالا و حفظ کیفیت تصویر تعادل وجود دارد. نسبت تراکم بالاتر ممکن است منجر به از دست رفتن قابل توجه جزئیات تصویر و مصنوعات بصری شود. برای ایجاد تعادل، مهم است که الزامات خاص برنامه را در نظر بگیریم. تنظیم پارامترهای فشرده‌سازی، استفاده از تکنیک‌های فشرده‌سازی تطبیقی یا استفاده از حالت‌های فشرده‌سازی مختلف (به عنوان مثال، پیشرونده یا سلسله مراتبی) می‌تواند به بهینه‌سازی مبادله بین نسبت فشرده‌سازی و کیفیت تصویر کمک کند.

- از دست دادن داده های اصلی: تکنیک های فشرده سازی با اتلاف برخی از اطلاعات تصویر را برای دستیابی به فشرده سازی بالاتر دور می اندازند. این بدان معنی است که داده های اصلی و غیر فشرده را نمی توان به طور کامل از نسخه فشرده بازیابی کرد. برای جلوگیری از این امر، می‌توان تکنیک‌های فشرده‌سازی بدون تلفات را در زمانی که حفظ تمام جزئیات تصویر حیاتی است، مانند تصویربرداری پزشکی یا اهداف آرشیوی، انتخاب کرد.

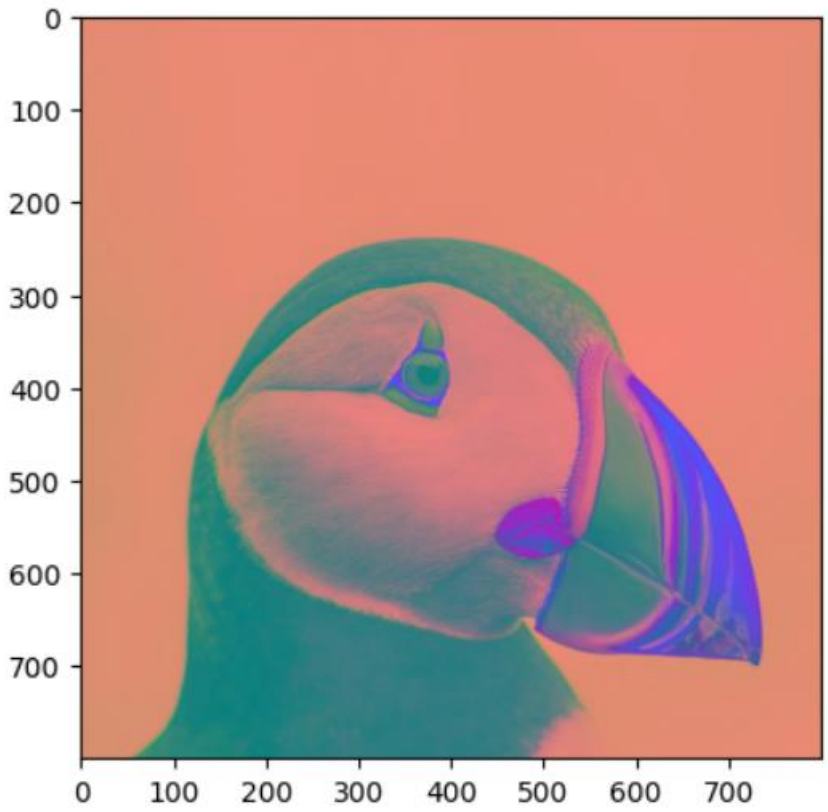
بخش دوم – تمرینات عملی)

۱- الگوریتم فشرده سازی هافمن

ابتدا تصویر را خوانده و به ycbcr تبدیل میکنیم:

```
•[60]: low_chroma_array = rgb2ycbcr(image_array)
       # convert BGR to YCrCb
       plt.imshow(low_chroma_array)
```

```
[60]: <matplotlib.image.AxesImage at 0x1856279e250>
```



سپس آن را به بلوک های ۸*۸ تقسیم کرده و تبدیل DCT را بر روی آن اعمال میکنیم:

[62]:
```
def perform_dct(image):
    height, width, channels = image.shape
    blocks = []

    for y in range(0, height, 8):
        for x in range(0, width, 8):
            block = image[y:y+8, x:x+8]
            dct_block = dct(dct(block.T, norm='ortho').T, norm='ortho')
            blocks.append(dct_block)

    return np.array(blocks)
```

[63]:
```
dct_coefficients = perform_dct(low_chroma_array)
```

[64]:
```
dct_coefficients
```

[64]:
```
array([[[[ 7.87919201e+02,  2.26250000e+02,  7.43338472e+01],
         [ 7.88327449e+02,  2.26750000e+02,  7.46225223e+01],
         [ 7.88327449e+02,  2.26750000e+02,  7.46225223e+01],
         ...,
         [ 7.88123325e+02,  2.27250000e+02,  7.57772228e+01],
         [ 7.88939821e+02,  2.29250000e+02,  7.63545731e+01],
         [ 7.87919201e+02,  2.29250000e+02,  7.51998726e+01]],

        [[-1.89021740e+00, -1.47788146e+00,  1.13405299e-01],
         [-1.36706447e+00, -8.37152602e-01,  4.83330280e-01],
         [-1.36706447e+00, -8.37152602e-01,  4.83330280e-01],
         ...,
         [ 6.83532237e-01,  8.37152602e-01,  4.83330280e-01],
         [ 1.17275591e-01,  1.53067252e+00,  8.29263658e-02],
         [-1.93332112e+00,  2.02106516e+00, -7.66458603e-01]],
```

در گام بعد ضرایب بدست آمده را کوانتایز میکنیم:

```
    quantization_matrix = np.transpose(quantization_matrix, (1, 2, 0))
    quantized_coefficients = np.round(dct_coefficients / quantization_matrix)
    return quantized_coefficients.astype(int)
```

```
quantization_matrix = np.array([
    [
        [16,  11,  10,  16,  24,  40,  51,  61],
        [12,  12,  14,  19,  26,  58,  60,  55],
        [14,  13,  16,  24,  40,  57,  69,  56],
        [14,  17,  22,  29,  51,  87,  80,  62],
        [18,  22,  37,  56,  68, 109, 103,  77],
        [24,  35,  55,  64,  81, 104, 113,  92],
        [49,  64,  78,  87, 103, 121, 120, 101],
        [72,  92,  95,  98, 112, 100, 103,  99]
    ],
    # Repeat the same matrix for chrominance (Cb and Cr) components
    [
        [17,  18,  24,  47,  99,  99,  99,  99],
        [18,  21,  26,  66,  99,  99,  99,  99],
        [24,  26,  56,  99,  99,  99,  99,  99],
        [47,  66,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99]
    ],
    # Repeat the same matrix for chrominance (Cb and Cr) components
    [
        [17,  18,  24,  47,  99,  99,  99,  99],
        [18,  21,  26,  66,  99,  99,  99,  99],
        [24,  26,  56,  99,  99,  99,  99,  99],
        [47,  66,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
```

```
quantized_coefficients = quantize_dct_coefficients(dct_coefficients, quantizati
quantized_coefficients
```

```
array([[[[49, 13,  4],
         [72, 13,  4],
         [79,  9,  3],
         ...,
         [20,  2,  1],
         [15,  2,  1],
         [13,  2,  1]],

        [[ 0,  0,  0],
         [ 0,  0,  0],
         [ 0,  0,  0],
         ...,
         [ 0,  0,  0],
         [ 0,  0,  0],
         [ 0,  0,  0]],

        [[ 0,  0,  0],
         [ 0,  0,  0],
         [ 0,  0,  0],
```

مشاهده میکنیم که بسیاری از ضرایب صفر میشوند.

در نهایت به کمک الگوریتم هافمن ضرایب را کد میکنیم:

[70]:
```
def encode_quantized_coefficients(huffman_tree, quantized_coefficients):
    encoded_string = ""

    for coefficient in quantized_coefficients.flatten():
        encoded_string += huffman_tree[coefficient]

    return encoded_string
```

[98]:
```
flattened_coefficients = quantized_coefficients.flatten()
coefficient_counts = Counter(flattened_coefficients)
huffman_tree = find_huffman(coefficient_counts)
encoded_image = encode_quantized_coefficients(huffman_tree, quantized_coefficie
```

[72]:
```
huffman_tree
```

[72]:
```
{0: '1',
 -1: '0000',
 1: '0111',
 2: '0101',
 5: '00110',
 13: '00100',
 -2: '010010',
 3: '000110',
 4: '001111',
 10: '000101',
 50: '001011',
 -3: '0011101',
 9: '0001001',
 12: '0001111',
 14: '0110011',
 15: '0010101',
 16: '0100010',
```

تابع زیر را برای محاسبه حجم اطلاعات پس از فشرده سازی پیاده سازی میکنیم:

[73]:
```python
import json
import numpy as np

def calculate_huffman_size(huffman_tree, encoded_string):
    """
    Calculates the size of the Huffman tree and the encoded string in bits.

    Args:
        huffman_tree: The Huffman tree as a dictionary.
        encoded_string: The encoded string.

    Returns:
        tuple: A tuple containing the size of the Huffman tree and the encoded
    """
    huffman_tree_str = json.dumps({int(k): v for k, v in huffman_tree.items()})
    huffman_tree_size_in_bits = len(huffman_tree_str) * 8

    encoded_string_size_in_bits = len(encoded_string)

    return huffman_tree_size_in_bits + encoded_string_size_in_bits
```

[74]:
```python
print(f'uncompressed: {calculate_huffman_size(huffman_tree, encoded_image) / (
```

```
uncompressed: 400.29625 KB
```

تا اینجا عکس فشرده شده.

حال توابعی را مینویسیم تا از روی اطلاعات مربوط به فشرده سازی عکس اصلی را بازیابی کنند:

تابع زیر با داشتن درخت هافمن یک عبارت کد شده را دیکود میکند:

Decode

```
def decode_huffman(binary_string, huffman_tree):
    """
    Decodes a binary string using a Huffman tree dictionary.
    :param binary_string: Binary string to be decoded
    :param huffman_tree: Huffman tree dictionary
    :return: Decoded string
    """

    decoded_string = []
    current_code = ""

    for bit in binary_string:
        current_code += bit

        if current_code in huffman_tree.values():
            symbol = list(huffman_tree.keys())[list(huffman_tree.values()).index(current_code)]
            decoded_string.append(symbol)
            current_code = ""

    return decoded_string
```

```
decoded_image = np.array(decode_huffman(encoded_image, huffman_tree)).reshape((10000, 8, 8, 3))
```

```
decoded_image
```

```
array([[[[49, 13,  4],
         [72, 13,  4],
         [79,  9,  3],
         ...,
```

پس از دیکود کردن ضرایب باید آن ها را دی کوانتایز کنیم:

[78]:
```python
def inverse_quantize(quantized_coefficients, quantization_matrix):
    """
    Performs inverse quantization on quantized coefficients using a quantization matrix.
    :param quantized_coefficients: Quantized coefficients (3D array)
    :param quantization_matrix: Quantization matrix (3D array)
    :return: Inverse quantized coefficients (3D array)
    """
    quantization_matrix = np.transpose(quantization_matrix, (1, 2, 0))
    inverse_quantized_coefficients = quantized_coefficients * quantization_matrix

    return inverse_quantized_coefficients
```

[79]:
```python
dequantized = inverse_quantize(decoded_image, quantization_matrix)
```

[80]:
```python
dequantized
```

[80]:
```
array([[[784, 221,  68],
        [792, 234,  72],
        [790, 216,  72],
        ...,
        [800, 198,  99],
        [765, 198,  99],
        [793, 198,  99]],

       [[  0,   0,   0],
        [  0,   0,   0],
        [  0,   0,   0],
        ...,
        [  0,   0,   0],
        [  0,   0,   0],
        [  0,   0,   0]],
```

در نهایت با انجام IDCT و سر هم کردن بلوک را تصویر بازیابی شده را میسازیم:

```python
def reconstruct_image(blocks, image_shape):
    """
    Reconstructs the original image from blocks of DCT coefficients.

    Args:
        blocks (ndarray): Blocks of DCT coefficients. Shape: (num_blocks, block_size, block_size, num_channels)
        image_shape (tuple): Shape of the original image (height, width, num_channels).

    Returns:
        ndarray: Reconstructed image.
    """
    height, width, num_channels = image_shape
    block_size = blocks.shape[1]

    num_blocks_h = height // block_size
    num_blocks_w = width // block_size

    # Initialize an array to store the reconstructed image
    reconstructed_image = np.zeros(image_shape)

    block_index = 0
    for y in range(0, height, block_size):
        for x in range(0, width, block_size):
            block = blocks[block_index]
            idct_block = idct(idct(block.T, norm='ortho').T, norm='ortho')
            reconstructed_image[y:y+block_size, x:x+block_size] = idct_block
            block_index += 1

    return reconstructed_image
```

```python
inverse_dct = reconstruct_image(dequantized, (800, 800, 3))
```

و با تبدیل عکس به RGB و صحیح کردن مقادیر درایه ها آن را نمایش میدهیم:

```
show_image(final_rgb)
```

خروجی کیفیت و حجم کمتری دارد.

مدار فشرده سازی در زیر نمایش داده شده:

```
huffman_tree, encoded_image, compression_ratio = encode("slider_puffin_before_mobile.jpg")
```

```
uncompressed: 1920.0 KB
compressed: 400.29625 KB
compression ratio: 4.796447630973311
```

```
output = decode(huffman_tree, encoded_image)
show_image(output)
```

بخش های امتیازی:

۲-ماتریس های مختلفی را برای کوانتزیشن امتحان میکنیم و میبینیم که کیفیت و نسبت فشرده سازی به صورت زیر تغییر میکند:

کیفیت: فرآیند کوانتیزه کردن خطاهای کوانتیزه کردن را معرفی می کند که می تواند منجر به کاهش کیفیت تصویر شود. ضریب کوانتیزاسیون بالاتر (مقادیر بزرگتر در ماتریس کوانتیزاسیون) منجر به کوانتیزه سازی تهاجمی تر و از دست دادن اطلاعات بیشتر و در نتیجه کیفیت تصویر پایین تر می شود. از سوی دیگر، ضریب کوانتیزاسیون کمتر

(مقادیر کوچکتر در ماتریس کوانتیزاسیون) جزئیات بیشتری را حفظ می کند و منجر به کیفیت تصویر بالاتر می شود. با این حال، فاکتورهای کوانتیزاسیون بسیار کم ممکن است منجر به اندازه فایل بزرگتر و فشرده سازی کمتر شود.

نسبت فشرده سازی: نسبت فشرده سازی نسبت اندازه داده های فشرده شده به اندازه داده های اصلی است. تغییر ماتریس کوانتیزاسیون بر نسبت فشرده سازی تأثیر می گذارد زیرا مستقیماً بر مقدار داده هایی که می توانند در طول فرآیند کوانتیزه کردن دور ریخته شوند تأثیر می گذارد. فاکتورهای کوانتیزاسیون بالاتر منجر به حذف داده های بیشتری می شود که منجر به نسبت تراکم بالاتر می شود. برعکس، فاکتورهای کوانتیزاسیون کمتر، اطلاعات بیشتری را حفظ می‌کنند و در نتیجه نسبت تراکم پایین‌تری دارند.

```
quantization_matrix = np.array([
    [
        [12,  16,  19,  22,  26,  27,  29,  34],
        [16,  16,  22,  24,  27,  29,  34,  37],
        [19,  22,  26,  27,  29,  34,  37,  38],
        [22,  24,  27,  29,  34,  37,  38,  39],
        [26,  27,  29,  32,  35,  39,  40,  44],
        [27,  29,  34,  37,  39,  40,  44,  48],
        [29,  34,  37,  38,  40,  44,  48,  58],
        [34,  37,  38,  39,  44,  48,  58,  69]
    ],
    # Repeat the same matrix for chrominance (Cb and Cr) components
    [
        [17,  18,  24,  47,  99,  99,  99,  99],
        [18,  21,  26,  66,  99,  99,  99,  99],
        [24,  26,  56,  99,  99,  99,  99,  99],
        [47,  66,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99]
    ],
    # Repeat the same matrix for chrominance (Cb and Cr) components
    [
        [17,  18,  24,  47,  99,  99,  99,  99],
        [18,  21,  26,  66,  99,  99,  99,  99],
        [24,  26,  56,  99,  99,  99,  99,  99],
        [47,  66,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99]
    ]
])
huffman_tree, encoded_image, compression_ratio = encode("slider_puffin_before_mobile.jpg")
output = decode(huffman_tree, encoded_image)
show_image(output)
```

```
uncompressed: 1920.0 KB
compressed: 400.909625 KB
compression ratio: 4.78910926620931
```

```
quantization_matrix = np.array([
    [
        [6,  4,  4,  6,  10,  16,  20,  24],
        [5,  5,  6,  8,  10,  23,  24,  22],
        [6,  5,  6,  10, 16,  23,  28,  22],
        [6,  7,  9,  12, 20,  35,  32,  25],
        [7,  9,  15, 22, 27,  44,  41,  31],
        [10, 14, 22, 26, 32,  42,  45,  37],
        [20, 26, 31, 35, 41,  48,  48,  40],
        [29, 37, 38, 39, 45,  40,  41,  40]
    ],
    # Repeat the same matrix for chrominance (Cb and Cr) components
    [
        [17,  18,  24,  47,  99,  99,  99,  99],
        [18,  21,  26,  66,  99,  99,  99,  99],
        [24,  26,  56,  99,  99,  99,  99,  99],
        [47,  66,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99]
    ],
    # Repeat the same matrix for chrominance (Cb and Cr) components
    [
        [17,  18,  24,  47,  99,  99,  99,  99],
        [18,  21,  26,  66,  99,  99,  99,  99],
        [24,  26,  56,  99,  99,  99,  99,  99],
        [47,  66,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99]
    ]
])
huffman_tree, encoded_image, compression_ratio = encode("slider_puffin_before_mobile.jpg")
output = decode(huffman_tree, encoded_image)
show_image(output)
```

```
uncompressed: 1920.0 KB
compressed: 430.0535 KB
compression ratio: 4.464560804644074
```

```
quantization_matrix = np.array([
    [
        [3,  5,  7,  9,  11,  13,  15,  17],
        [5,  7,  9,  11, 13,  15,  17,  19],
        [7,  9,  11, 13,  15,  17,  19,  21],
        [9,  11, 13,  15,  17,  19,  21,  23],
        [11, 13,  15,  17,  19,  21,  23,  25],
        [13,  15,  17,  19,  21,  23,  25,  27],
        [15,  17,  19,  21,  23,  25,  27,  29],
        [17,  19,  21,  23,  25,  27,  29,  31]
    ],
    # Repeat the same matrix for chrominance (Cb and Cr) components
    [
        [17,  18,  24,  47,  99,  99,  99,  99],
        [18,  21,  26,  66,  99,  99,  99,  99],
        [24,  26,  56,  99,  99,  99,  99,  99],
        [47,  66,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99]
    ],
    # Repeat the same matrix for chrominance (Cb and Cr) components
    [
        [17,  18,  24,  47,  99,  99,  99,  99],
        [18,  21,  26,  66,  99,  99,  99,  99],
        [24,  26,  56,  99,  99,  99,  99,  99],
        [47,  66,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99],
        [99,  99,  99,  99,  99,  99,  99,  99]
    ]
])
huffman_tree, encoded_image, compression_ratio = encode("slider_puffin_before_mobile.jpg")
output = decode(huffman_tree, encoded_image)
show_image(output)
```

```
uncompressed: 1920.0 KB
compressed: 433.242875 KB
compression ratio: 4.431694346964159
```

۴-این بار از HSV استفاده میکنیم:

```python
def rgb2hsv(image):
    hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    return hsv_image
```

```python
def hsv2rgb(image):
    # Convert image to the appropriate data type
    image = image.astype(np.uint8)

    # Convert HSV to RGB
    rgb_image = cv2.cvtColor(image, cv2.COLOR_HSV2BGR)

    return rgb_image
```

```python
coefficient_counts, encoded_image, compression_ratio = encode("slider_puffin_before_mobile.jpg")
output = decode(coefficient_counts, encoded_image, )
show_image(output)
```

```
uncompressed: 1920.0 KB
compressed: 491.51525 KB
compression ratio: 3.90628774997317
```

۲- الگوریتم K-means

الگوریتم را به صورت زیر پیاده سازی میکنیم:

```python
def k_means_compression(image, k, max_iters=100, threshold=1e-5):
    """
    Performs k-means compression on the input image.

    Args:
        image: The input image as a NumPy array with shape (width, height, 3).
        k: The number of clusters/centroids.
        max_iterations: Maximum number of iterations to run the algorithm.
        tolerance: The convergence tolerance for centroid change.

    Returns:
        compressed_image: The compressed image as a NumPy array with the same shape as the input image.

    """
    # Reshape the image into a flattened array of pixels
    pixels = image.reshape(-1, 3)

    # Step 1: K-means++ Initialization
    centroids = kmeans_plusplus_initialization(pixels, k)

    # Step 2: K-means Clustering
    while True:
        # Step 3: Assign pixels to nearest centroid
        labels = assign_labels(pixels, centroids)

        # Step 4: Update centroids
        new_centroids = update_centroids(pixels, labels, k)

        # Step 5: Check convergence
        if np.linalg.norm(new_centroids - centroids) < threshold:
            break

        centroids = new_centroids

    # Step 6: Replace pixel values with nearest centroid
    compressed_image = centroids[labels]

    # Reshape the compressed image back to its original shape
    compressed_image = compressed_image.reshape(image.shape)

    return compressed_image
```

و برای تعیین مراکز اولیه از K-means++ به صورت زیر استفاده میکنیم:

```python
def kmeans_plusplus_initialization(pixels, k):
    """
    This method improves the chances of selecting good initial centroids by considering the distance between points.
    It starts by randomly selecting one centroid, and then selects subsequent centroids with a probability
    proportional to their distance from the existing centroids.
    """
    centroids = []

    # Choose the first centroid randomly
    centroid = pixels[np.random.choice(pixels.shape[0])]
    centroids.append(centroid)

    # Calculate the distances to the existing centroids
    distances = np.linalg.norm(pixels - centroid, axis=1)

    for _ in range(1, k):
        # Choose the next centroid with probability proportional to squared distance
        probabilities = distances ** 2
        probabilities /= np.sum(probabilities)

        centroid = pixels[np.random.choice(pixels.shape[0], p=probabilities)]
        centroids.append(centroid)

        # Update the distances to the existing centroids
        new_distances = np.linalg.norm(pixels - centroid, axis=1)
        distances = np.minimum(distances, new_distances)

    return np.array(centroids)
```

خروجی برای K=۱۵ به صورت زیر خواهد بود:

```python
result = k_means_compression(image_array, 15)
```

```python
show_image(result)
```

۶-در این بخش ضرایب DCT را به تصویر میکشیم، برای مثال در شکل زیر ضریب ۳۳۳۳ نشان داده شده است:

visulization

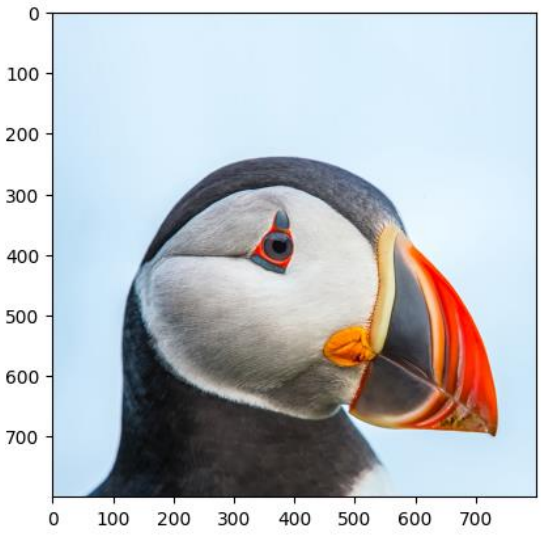
```
import matplotlib.pyplot as plt
import numpy as np

def visualize_dct_coefficient(dct_coefficients, coefficient_index):
    # Select the DCT coefficient from the matrix
    coefficient = dct_coefficients[coefficient_index]

    # Rescale the coefficient values to the range [0, 255]
    rescaled_coefficient = (coefficient - np.min(coefficient)) / (np.max(coefficient) - np.min(coefficient))

    # Create a grayscale image from the integer coefficient
    plt.imshow(rescaled_coefficient)
    plt.axis('off')
    plt.show()
```

```
visualize_dct_coefficient(dct_coefficients,3333)
```

۳-کدگذاری حسابی را به صورت زیر پیاده میکنیم و میبینیم که نسبت فشرده سازی بیشتر شده اما کیفیت عکس نیز بالا میرود:

```python
def get_cumulative_probabilities(probabilities):
    cumulative_probs = {}
    cum_prob = 0.0
    for symbol, prob in probabilities.items():
        cum_prob += prob
        cumulative_probs[symbol] = cum_prob
    return cumulative_probs


def arithmetic_encode(sequence, cumulative_probs):
    # Get the total number of symbols in the sequence
    num_symbols = len(sequence)

    # Initialize the interval and range
    interval = 0.0
    range_width = 1.0

    # Encode the symbols in the sequence
    for symbol in sequence:
        symbol_prob = cumulative_probs[symbol]
        interval += range_width * symbol_prob
        range_width *= cumulative_probs.get(symbol + 1, 0) - symbol_prob

    # Return the encoded interval
    return interval
```

```python
def arithmetic_decode(encoded_value, probabilities, num_symbols):
    # Create cumulative probability distribution
    cumulative_probs = [0] + list(np.cumsum(probabilities))

    # Initialize interval [0, 1) and range [0, 1)
    interval = encoded_value
    range_width = 1

    sequence = []

    for _ in range(num_symbols):
        for symbol, symbol_prob in enumerate(probabilities):
            symbol_range_start = cumulative_probs[symbol]
            symbol_range_end = cumulative_probs[symbol + 1]

            if symbol_range_start <= interval < symbol_range_end:
                # Output the symbol
                sequence.append(symbol)

                # Update interval and range
                interval = (interval - symbol_range_start) / range_width
                range_width *= symbol_prob
                break

    return sequence
```



سوالات:

- **چرا این الگوریتم lossy است؟**

وشه بندی K-means اغلب برای کوانتیزه کردن رنگ در فشرده سازی تصویر استفاده می شود. رنگ‌های مشابه را با هم ترکیب می‌کند و آنها را با رنگ‌های نماینده خوشه‌ها جایگزین می‌کند. تعداد خوشه ها، k، سطح کاهش رنگ را تعیین می کند. در نتیجه، برخی از اطلاعات رنگ در این فرآیند از بین می رود. پالت رنگ کاهش یافته ممکن است به دقت تصویر اصلی را نشان ندهد، که منجر به تقریب رنگ و از بین رفتن احتمالی جزئیات رنگی می شود. همچنین ممکن است به طور موثر جزئیات پیچیده یا الگوهای بافت را در تصویر ثبت نکند. جزئیات ظریف، مانند بافت، گرادیان، یا تغییرات ظریف، ممکن است در طول فرآیند فشرده سازی از بین بروند یا تقریبی پیدا کنند که منجر به از دست دادن وفاداری بصری می شود.

خوشه‌بندی K-means هر پیکسل را به نزدیک‌ترین مرکز خوشه‌ای اختصاص می‌دهد که خطاهای کوانتیزاسیون را معرفی می‌کند. مقادیر پیکسل به رنگ نماینده خوشه تقریبی می‌شوند و در نتیجه تفاوت‌هایی بین مقادیر پیکسل اصلی و نمایش فشرده ایجاد می‌شود. این خطاهای کوانتیزاسیون می تواند منجر به مصنوعات قابل مشاهده شود، مانند لبه های بلوک یا ناهموار، به ویژه در مناطقی با انتقال های تیز یا اجزای فرکانس بالا.

- **تعداد دسته ها چه تاثیری بر قدرت فشرده سازی و میزان lossy و سرعت اجرا دارد؟**

- توان فشرده سازی: تعداد خوشه ها به طور مستقیم بر توان فشرده سازی یا بازده فشرده سازی k-means تأثیر می گذارد. افزایش تعداد خوشه ها امکان نمایش دقیق تری از رنگ ها در تصویر را فراهم می کند و در نتیجه کیفیت رنگ بهتری را به همراه دارد. تعداد بیشتری از خوشه ها می توانند تنوع رنگ بیشتری را ثبت کنند که منجر به سطح بالاتری از قدرت فشرده سازی می شود. با این حال، افزایش تعداد خوشه ها همچنین مقدار داده های مورد نیاز برای نشان دادن مرکزهای خوشه را افزایش می دهد، که می تواند بر نسبت فشرده سازی کلی تأثیر بگذارد.

- نرخ تلفات: نرخ تلفات به مقدار اطلاعات یا کیفیت از دست رفته در طول فشرده سازی اشاره دارد. در فشرده‌سازی k-means، تعداد بیشتری از خوشه‌ها می‌توانند به طور بالقوه نرخ تلفات را کاهش دهند. با خوشه‌های بیشتر، کوانتیزه کردن رنگ‌ها ظریف‌تر می‌شود و در نتیجه اطلاعات رنگ کاهش می‌یابد و نمایش رنگی دقیق‌تر می‌شود. بنابراین، افزایش تعداد خوشه ها منجر به کاهش نرخ تلفات و بهبود کیفیت تصویر فشرده می شود. با این حال، یافتن تعادل بسیار مهم است، زیرا افزایش بیش از حد تعداد خوشه‌ها می‌تواند منجر به تطبیق بیش از حد و از بین رفتن جزئیات تصویر شود.

- سرعت اجرا: سرعت اجرای فشرده سازی k-means تحت تأثیر تعداد خوشه ها قرار می گیرد. با افزایش تعداد خوشه ها، پیچیدگی محاسباتی الگوریتم خوشه بندی نیز افزایش می یابد. تخصیص پیکسل ها به تعداد بیشتری از خوشه ها به محاسبات بیشتری نیاز دارد و در نتیجه زمان اجرا طولانی تر می شود. بنابراین، افزایش تعداد خوشه‌ها در فشرده‌سازی k-means عموماً منجر به کاهش سرعت فشرده‌سازی و کاهش فشار می‌شود. از سوی دیگر، استفاده از خوشه های کمتر فرآیند را سرعت می بخشد اما ممکن است منجر به کیفیت فشرده سازی پایین تر شود.